Solaris VMEbus Driver
Programmer's Guide

P/N 204936 Edition 9.0
January 2000

A SOLECTRON SUBSIDIARY

World Wide Web: www.forcecomputers.com
24-hour access to on-line manuals, driver updates, and application notes is provided via SMART, our SolutionsPLUS customer support program that provides current technical and services information.

Headquarters

The Americas

Force Computers Inc.
5799 Fontanoso Way
San Jose, CA 95138-1015
U.S.A.

Tel.: +1 (408) 369-6000
Fax: +1 (408) 371-3382
Email support@fci.com

Europe

Force Computers GmbH
Prof.-Messerschmitt-Str. 1
D-85579 Neubiberg/München
Germany

Tel.: +49 (89) 608 14-0
Fax: +49 (89) 609 77 93
Email support@force.de

Asia

Force Computers Japan KK
Shiba Daimon MF Building 4F
2-1-16 Shiba Daimon
Minato-ku, Tokyo 105-0012
Japan
Tel.: +81 (03) 3437 3948
Fax: +81 (03) 3437 3968
Email smiyagawa@fci.com

Table of Contents

204936 9 – 0 January 2000

204936 9 – 0 January 2000

List of Tables and Figures

Page **Tab./Fig.**

2041936 9 – 0 January 2000

Using This Manual

This section does not provide information on the product but on common features of the manual itself:

- its structure
- special layout conventions
- and related documents

Audience of the Manual

This *Programmer's Guide* is intended for software developers writing applications and drivers for VMEbus hardware under Solaris x running on a Force Computers CPU.

- The standard UNIX system calls and the VUI will be of importance to application programmers and are primarily covered in section 4 "Application Programmer's Guide" on page 27,
- whereas the DDI and VDI are important to VMEbus leaf driver developers who will be called device driver developers in this user's manual. The DDI and VDI interfaces are primarily covered in section 5 "Device Driver Developer's Guide" on page 99.

Throughout this manual it is assumed that you are generally familiar with Solaris x and have a working knowledge of Solaris x device drivers and VMEbus device drivers, in particular. Since this manual refers to the following documents, it is recommended that you have them available to consult (e.g. via `http://docs.sun.com`):

- Solaris x Driver Developer AnswerBook
- Solaris x Software Developer AnswerBook
- Solaris x Writing Device Drivers
- Solaris x DDI and DKI Kernel Functions

Note: **This *Programmer's Guide* describes the standard Solaris VMEbus Driver package. Note that the *Release Notes* of the respective version of the Solaris VMEbus Driver describe hardware and software dependencies as well as limitations which may apply to a specific CPU board for the release under consideration.**
To understand a CPU board's VMEbus behavior the technical reference manual of the CPU board's VMEbus interface chip (e.g. the FGA-5000) and of the CPU board itself (e.g. the SPARC/CPU-20VT) are recommended.

Overview of the Manual

This *Programmer's Guide* provides a comprehensive software guide to the Force Computers VMEbus driver suite.

Note: **Please take a moment to examine the "Table of Contents" to see how this documentation is structured. This will be of value to you when looking for information in the future.**

It includes

- an overview of the safety notes: see section 1 "Safety Notes" on page 1
- a brief overview of the product and of changes as compared to old-style VMEbus drivers: see section 2 "Introduction" on page 5
- the installation instructions for the driver suite: see section 3 "Installation and Configuration Guide" on page 15. It includes the default configuration and initialization.
- a detailed description for user application programmers: see section 4 "Application Programmer's Guide" on page 27
- a detailed description for device driver developers: see section 5 "Device Driver Developer's Guide" on page 99
- a description of the extended bustypes concept – called bus properties – and the bus properties defined: see section 6 "VME Bus Properties" on page 159
- a description of the system messages and possible causes to them: see section 7 "System Messages" on page 165

204936 9 – 0 January 2000

Glossary

The following terminology is used throughout this manual:

2eVME	2 edge VME
AM	Address modifier
AS	Address space
BBSY	Bus busy signal
BCLR	Bus clear
Bit	A bit is either “set” (bit = 1) or cleared (bit = 0).
BLT	VME block transfer (32 Bit)
BRL	Bus request level
BRM	Bus request mode
BREL	Bus release mode (➤RAT, ➤ROC, ➤ROR, ➤RWD)
DMA	Direct memory access
DVMA	Direct virtual memory access
DMAC	DMA controller
FDMA	Fast direct memory access: when writing of fast DMA, the software method to increase the actual transfer speed is meant. It is not a hardware feature.
IACK	Interrupt acknowledge
IRQ	Interrupt request
Mailbox	An address location on the VMEbus which triggers a local interrupt when accessed.
MBLT	VME multiplexed block transfer (64 Bit)
Bustype	This is a set of properties which describes the way in which the VMEbus is accessed, i.e. the address and data modes, the transfer mode and other special conditions like write posting, data pre-fetch, etc.

Local bus	The I/O bus of the system to which the VME interface is attached (e.g. SBus or PCI bus).
Pfn	Page frame number. A physical on-board address divided by the MMU page size.
PRI	Prioritized (arbiter mode)
PRIRR	Prioritized round robin (arbiter mode)
RMW	Read-Modify-Write cycle
RAT	Release on time-out
ROC	Release on ➤BCLR
ROR	Release on request
RORA	Release on Register Accesses Interrupters (RORA): When a VMEbus interrupt request has been recognized, the CPU performs a write access to a control register in the interrupting device. In turn, the interrupting device removes its interrupt request.
RR	Round robin (arbiter mode)
RSW	Register slave window: This is a special slave window which enables access to the VME interface registers from VMEbus side.
RWD	Release when done
Semaphore	A VME slave address provided by the local VME board which can be read and set in one atomic operation.
SGL	Single level (arbiter mode)
UAT	Unaligned transfer
VDI	VMEbus driver interface: This is one of the 2 parts of the interface between the VMEbus nexus driver and the VMEbus leaf drivers. The other one is the standard Solaris DDI. All functions starting with `vdi_` are collectively called VDI functions.
VUI	VME user application interface: This is one of the 2 parts of the interface between a user application and the VMEbus leaf drivers. The other one are standard UNIX system calls. All functions starting with `vui_` are collectively called VUI functions.

204936 9 – 0 January 2000

Publication History of the Manual

Table a History of manual editions

Ed.	Date	Description
1	July 1996	First print
2	August 1996	Added description of fast DMA driver `vmefdma` and `vui_reg_base_set/~get`
3	October 1996	Added bustypes, message, and `vui_mbox_info()` descriptions
4	June 1997	Started delivery of this manual edition with software release 2.1 • Added new bus properties (former term: bus types): `VME_BT_…` …`_PAMC1`, …`_PAMC2`, …`_CRCSR`, …`_A40`. • Support of all VMEbus AM codes. • Support of new devices: `vmedma32te`, `vmefdma32te`, `vmecrcsr16`, `vmecrcsr32`, `vmepam1d16`, `vmepam1d32`, `vmepam2d16`, and `vmepam2d32.` • Enhanced fault handling: a `siginfo` structure is passed to the process where possible. • New configuration flag `vme_event_warn`. • New functions: – `vui_…` and `vdi_…` `error_info()`, – `vdi_map…` …`_abs()` and …`_regspec()` – `vdi_intr_acknowledge()` and …`_generate()` – `vdi_virq…` …`_trigger()` and …`_ackwait()` • Programmable AM codes • Support for CR/CSR address space • VME shared memory can be allocated at fixed VME addresses. • Thoroughly revised examples • Revised terminology: to avoid misunderstandings, the FORCE COMPUTERS' extension of the bus type concept now uses the term "bus property" instead of "bus type". • Revised description of ERRORS within function descriptions: only the errors defined by the Solaris VMEbus Driver package are described in this manual, for all others the respective man pages are referenced.

Table a History of manual editions

Ed.	Date	Description
5	March 1998	Started delivery of this manual edition with software release 2.3: • Revised the manual to also fit for UltraSPARC technology where the SBus is replaced by the PCI bus as local bus • `/platform/`*`arch`*`/kernel/drv` documented as additional installation directory • `VME_BT_USER` replaced by `VME_BT_NPRV`, but `VME_BT_USER` is still available • Revised error values for VUI • Extended UNIX system calls for `vmedvma` driver • Added `SMEM_DONTMAP` flag for `vui_slave_map()`, `…_unmap()` for `vmedvma` driver • Revised examples • `ioctl_mbox_info_t` structure instead of `ioctl_mbox_t` for `vui_mbox_info()` • New functions: – `vdi_dmac_alloc_handle()` – `vdi_mbox_iblock_cookie()`, `vdi_mbox_hilevel()` – `vui_nsysfail_wait()`, `~signal()`. • Updated description of mailbox interrupt levels • Revised `vmewin` and `vmewin`*`X`* syntax which control master window allocation • Added configuration option of the `vmectl` driver to control ACFAIL and SYSFAIL handling related to the function groups – `vui_acfail_wait()`, `~signal()` – and `vui_(n)sysfail_wait()`, `~signal()`. • Added flag `IMM_CALLBACK` for `vdi_event_setup()`.

204936 9 – 0 January 2000

Table a History of manual editions

Ed.	Date	Description
6.0	December 1998	Started delivery of this manual edition with software release 2.4: • New VDI functions: – `vdi_attach()` – `vdi_reg_space()` • New VUI function: – `vui_mbox_control()` • Updated interface description for – vui_mbox_set() – vui_mbox_wait() – vui_arb_mode_set() • Updated the description for vmedvma driver regarding the new features for mapping shared memory buffers to several processes. • Updated description of /`dev` entries for `vmeplus`, `vmedma` and `vmefdma` drivers. • Updated configuration section of vmeplus driver regarding the option to generate fixed-width VME accesses.
7.0	October 1999	• Changed syntax of `vdi_info()` • Removed return values for `vdi_success` • Added section “safety notes” • Editorial changes
8.0	November 1999	• Editorial changes
9.0	January 2000	• Changed manual type from Instruction Set to Programmer’s Guide

Fonts, Notations and Conventions

Table b **Fonts, notations and conventions**

Notation	Description
`0000.0000`$_{16}$	Typical notation for hexadecimal numbers (digits are 0 through F), e.g. used for addresses and offsets. Note the dot marking the 4th (to its right) and 5th (to its left) digit.
`0000`$_{8}$	Same for octal numbers (digits are 0 through 7)
`0000`$_{2}$	Same for binary numbers (digits are 0 and 1)
`Program`	Typical character format used for names, values, and the like that should be used typing literally the same word. Also used for on-screen-output.
Variable	Typical character format for words that represent a part of a command, a programming statement, or the like and that will be replaced by an applicable value when actually applied.
…set a flag…	means: set the flag to `1`.
…clear a flag…	means: set the flag to `0`.

Icons for Ease of Use: Safety Notes and Tips & Tricks

The following 3 types of safety notes appear in this manual. Be sure to always read and follow the safety notes of a section first – before acting as documented in the other parts of the section.

Danger

Dangerous situation: serious injuries to people or severe damage to objects.

Caution

Possibly dangerous situation: slight injuries to people or damage to objects possible.

Note: **No danger encountered. Pay attention to important information marked using this layout.**

204936 9 – 0 January 2000

1 Safety Notes

This section provides safety precautions to follow when using the Solaris VMEbus Driver. For your protection, follow all warnings and instructions found in the following text.

General

This *Programmer's Guide* provides the necessary information to handle the Solaris VMEbus Driver. As the product is complex and its usage manifold, we do not guarantee that the given information is complete. In case you need additional information, ask your Force Computers representative.

Application Programming

The handling capabilities for VMEbus write errors differ significantly depending on the type of hardware architecture used. The default reaction is therefore very conservative. Refer to the *Release Notes* for information on whether the behavior can be modified for the hardware under consideration.

Some VME leaf drivers may not be loaded on hardware that lacks the corresponding hardware features. For example, the `vmedma` and `vmefdma` drivers cannot be installed on S4 based boards, since the S4 SBus-to-VME bridge has no DMA controller.

It is not possible to set the transfer modes for individual device nodes or processes. At the moment when the transfer mode is set up, it is valid for the whole driver instance. For the `vmeplus` driver this means the following: if the transfer mode is set up for example for `/dev/vme32d32`, it is valid for all `/dev/vme`*`xx`*`d`*`yy`* devices and for all other processes using these devices. However, existing mappings will not be affected.

Some hardware needs properly aligned buffer and/or VMEbus addresses (refer to the *Release Notes* if this is true for the CPU board you use). To allocate the DMA buffer, it is recommended to use the VUI function `vui_dma_malloc()` which allocates properly aligned memory. For VMEbus addresses, it is safe to use page-aligned start addresses and sizes.

Depending on the system architecture, Solaris might not give the allocated memory back for normal use. Refer to the *Release Notes* for further information on allocating shared and DMA memory for the CPU board under consideration.

Before it is possible to read or write DMA memory via the `vmefdma` driver, it is necessary to allocate an I/O buffer via `vui_fdma_malloc()`. Use the resulting `ioaddr` returned by `vui_fdma_malloc()` as `buf` argument for `read()` or `write()` accesses.

Never change the `vmedvma` configuration file.

The generation of interrupts is hardware dependent. Therefore, refer to the *Release Notes* whether this feature is supported on the CPU board under consideration.

Enabling or disabling the SYSRESET output and input signal is switch-selectable. Therefore, check the CPU board's switch setting to ensure proper operation.

It may be that error events are dropped when using the flag `VME_SLEEP`. This is the case when an error occurs in the time between issuing one of the above function calls and actually waiting for an error event. To prevent such problems, the application programmer should set a timeout which interrupts the wait state from time to time and then check the error counters.

Device Driver

When changing a value for a programmable AM code in `VME.conf`, the `vmeplus.conf` has to be updated as well so that the bus property for the corresponding `reg` property reflects the new value in the `VME.conf` file.

The `VME_BT_PAMC1` and `VME_BT_PAMC2` bus properties do not define the address space size, which means that a `VME_BT_PAMC`*x* bus property literal must always be used in combination with a bus property specifying an address space (`VME_BT_Axx`).

The VME nexus driver does not attempt to perform an IACK cycle itself for interrupt levels at which such a non-vectored ISR is installed. However, hardware may require this. Therefore, the device driver developer must use `vdi_intr_acknowledge()` to obtain the interrupt vector, even if the vector is not used.

It is recommended to set the `IMM_CALLBACK` flag, because there is no other way to request the current status of the ACFAIL and SYSFAIL lines.

2049369 -- 0 January 2000

As of Solaris VMEbus Driver release 2.1 the `vdi_map_abs()` function is supported. It is strongly recommended to use `vdi_map_abs()`, instead of `vdi_map()`.

Depending on the hardware architecture, shared memory might be allocated non-cached. Once non-cached memory has been allocated by `vdi_smem_alloc()`, it may no longer be available for normal use by the virtual memory system. This is because Solaris removes memory from the free list once it has been set to non-cached. However, the memory will be re-used for future slave memory requests.

204936 9 -- 0 January 2000

2 Introduction

The Solaris VMEbus Driver package is a Solaris software extension to provide access to VMEbus devices for device drivers and user applications. It is intended for Solaris 2.5 or higher.
It logically consists of 2 main parts allowing for hardware and software independence:

Leaf drivers, device drivers

- the (VMEbus, SBus, PCI) leaf drivers: The term leaf driver refers to a device driver that accesses logically or physically existent devices on an I/O bus, and implements the functions defined for the device, such as transferring data to or from the device or accessing device registers.

  There are several leaf drivers available in the Solaris VMEbus Driver package, providing an application interface to the various functions of the VMEbus bridge (like VMEbus master accesses, VMEbus slave windows, operating the DMA controller, etc.). The Force Computers leaf drivers are shipped as binaries and source code, so they can be used as sample leaf drivers and extended on demand.

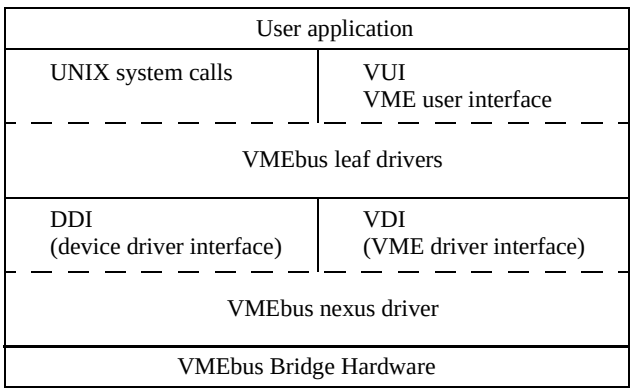
  The following leaf drivers are included in the Solaris VMEbus Driver package: `vmeplus`, `vmectl`, `vmedma`, `vmefdma`, and `vmedvma`. In this manual these leaf drivers are meant when writing about leaf drivers in general.

VMEbus nexus driver

- the VMEbus nexus driver: a nexus driver is a bus driver which interfaces leaf drivers to a specific I/O bus. The VMEbus nexus driver provides the low-level kernel integration of the VMEbus. It supports customer-specific VMEbus leaf drivers developed using the Solaris VMEbus Driver package. E.g., it implements auto-configuration, interrupt handling and memory mapping.

  In this manual the VME nexus driver is meant when writing about nexus drivers in general.

Figure 1 Architecture of the Solaris VMEbus Driver package



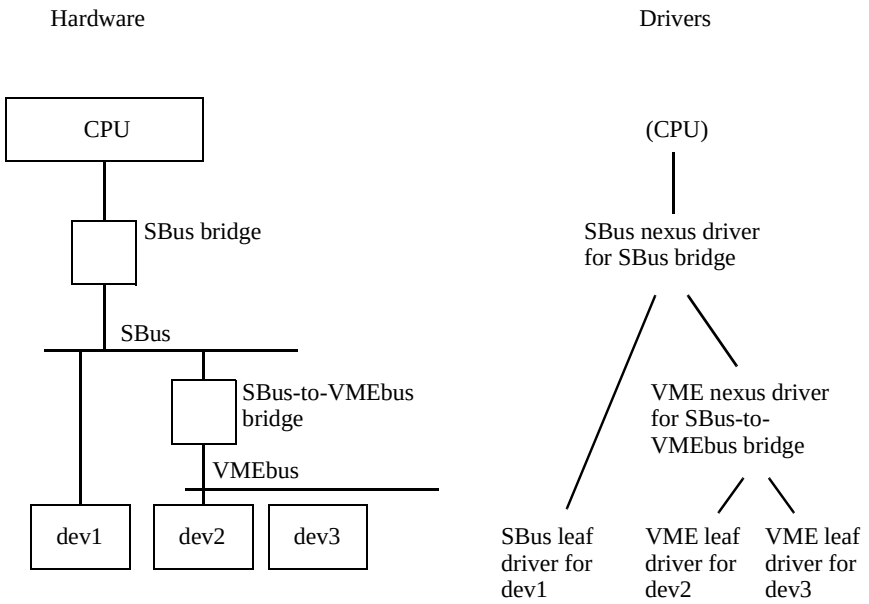
Access hierarchy

The VMEbus nexus driver provides the bus driver for the VMEbus which is used to access a VMEbus device. The device-specific VMEbus leaf driver handles the device itself. The following figure schematically shows the access hierarchy: if a device is accessed, the device's leaf driver (dev2) calls the respective bus driver (VME) which may well call another bus driver (SBus), as is the case for the dev2 device in the figure shown.

Example

When dev2 is accessed, the dev2 leaf driver is called to do the access. For this purpose, it calls the nexus driver for the SBus-to-VMEbus bridge which then itself calls another nexus driver – the one for the SBus bridge, and so on.

Figure 2 Sample device access hierarchy



VMEbus leaf drivers

In more detail, every device-specific VMEbus leaf driver sets up the device registers necessary for a particular function and supplies routines to establish mappings, transfer data, handle interrupts, access the device's registers and other device-specific routines.
The routines provide user applications with the access to the VMEbus functions via device nodes which are special files in the `/dev` directory:

– system calls

- either by the use of standard UNIX system calls such as `open()`, `read()`, `write()`, `mmap()`, etc. Called for VMEbus devices these system calls allow for data transfers to and from the VMEbus devices.

204936 9 – 0 January 2000

– VUI

- or by calling many other VMEbus specific functions which are provided by the VUI, the standard Force Computers VME user interface. The VUI provides, for example, functions waiting for external events or disabling the receipt of interrupts.

The functionality of the VMEbus leaf drivers and of the VUI depends on the VMEbus nexus driver and its interfaces. Only the functions supported by these interfaces also are supported by the VMEbus leaf driver and via the VUI.
There are different leaf drivers for different accesses and functions available. Each leaf driver supports his own list of devices:

- `vmeplus` replaces Sun's `vmemem` driver, it contains all the code necessary to access VMEbus memory. Furthermore, it provides a functionality to handle VMEbus interrupts and forward them as signal to the calling process. `vmeplus` is a standard character device driver. The driver provides access to the devices `/dev/vme`*xx*`d`*yy*
- `vmectl` provides various control functions to program the VMEbus bridge device. Furthermore, it supports handling of various Hardware signals like `abort`, `sysfail`, or `acfail`. The driver provides access to the device `/dev/vmectl`.
- `vmedma` utilizes on-board VME DMA controller (DMAC), it contains all the code necessary to initiate a transfer from user space to VMEbus memory and vice versa. The driver provides access to the `/dev/vmedma`*…* devices.
- `vmefdma` (the fast DMA driver) has basically the same functionality as `vmedma`. By reducing the software overhead for initiating a transfer it may be significantly faster than `vmedma`, with the disadvantage of a software interface which is not as flexible and convenient. The driver provides access to the devices `/dev/vmefdma`*….*
- `vmedvma` allows a process to allocate and map on-board memory so that it can be accessed both from the VMEbus and processes running locally. The driver provides access to the devices `/dev/vmedvma`*….*

VMEbus nexus driver

Just as with the VMEbus leaf drivers, the nexus driver provides 2 interfaces for the on-top software layer (which are VMEbus leaf drivers), the DDI and the VDI:

- The standard Solaris DDI (device driver interface) is described in the Solaris DDI documentation. It only provides a small segment out of the full range of VME functions.
- The Force Computers specific VDI (VME driver interface) enhances the DDI and provides, for example, DMA and VME control functions.

The VMEbus nexus driver and the VDI are operating system and hardware dependent, but both always provide the same interfaces to the VMEbus leaf drivers.

2.1 Software Interface Features

The following feature list may be limited due to hardware capabilities of the CPU board on which the Solaris VMEbus Driver package is installed. The limitations are described in the package's *Release Notes*.

VME master accesses	Address modes	A16, A24, A32, CR/CSR, 2 programmable AM codes
	Single transfer sizes	D8, D16, D32
	Other transfer modes	BLT, MBLT, 2eVME
	Access mode	supervisory/non-privileged, program/data
	RMW cycles	
VME slave accesses	Address modes	A16, A24, A32
	Single transfer sizes	D8, D16, D32
	Other transfer modes	BLT, MBLT, 2eVME
	Access mode	supervisory/non-privileged, program/data
	Slave memory	allocate, map to VMEbus, to kernel and to process
	Write posting	enable/disable
	Enable VME access to register set and set VME address	
DMA controller	Queue and execute DMA controller requests	
	DMA buffer	allocate and map to kernel
IRQs and signals	VMEbus IRQ(s)	receive, trigger, forward as signal
	SYSFAIL	react on, assert, clear
	ACFAIL, ABORT	react on
	SYSRESET	trigger
VME requester	Bus request	Levels 0…3, fair and demand mode
	Bus release	ROR, RWD, ROC, RAT
VME arbiter	Arbitration mode	Single level, round robin, priority, priority round robin

2049369 – 0 January 2000

Mailboxes	Mailbox access notification
	Request/set addresses and VME access properties for Mailboxes
Miscellaneous	Request CPU information (name, revision) and VME interface information (name, type, interface bus, revision)
	VMEbus error information.

2.2 Comparing the Old-Style to the New-Style Driver

Notes for application programmers

The following 2 changes should be noticed by application programmers:

- one concerns the change of device names
- the other concerns the use of `ioctl()`.

For more information on the new-style driver for application programmers see section 3 “Application Programmer’s Guide” on page 23.

Notes for device driver developers

The following should be noticed by device driver developers:

- There are no changes affecting the standard Solaris DDI.
- Changes have been made to the Force Computers specific interface between the VMEbus nexus and the VMEbus leaf driver. This interface is now called the VME Driver Interface – VDI. The changes affect the `vme_`*`xxx`*`()` functions.
- The `ioctl()` requests have also been changed.

For more information on the new-style driver for device driver developers see section 4 “Device Driver Developer’s Guide” on page 105.

Notes related to existing leaf drivers

The following list summarizes necessary changes to existing leaf drivers:

- Exclusively DDI-based leaf drivers run with the new-style driver without change.
- Leaf drivers which use the `vme_`*`xxx`*`()` functions have to use the respective `vdi_`*`xxx`*`()` functions, instead.

2.2.1 `vme_`*`xxx`*`()` Functions

Changes have been made to the 2 old-style driver functions named `vme_`*`xxx`*`()`. They are no longer available:

- `vme_dma_init()` is no longer necessary
- and instead of using `vme_dma_start()` use `vdi_dma_start()`.

Concerning functionality (not call syntax) the VDI provides a superset of the old-style `vme_xxx()` functions. Addtionally, contrary to the old-style `vme_xxx()` functions the VDI is hardware independent.

2.2.2 Device Names

The device names for `block` and `mblock` devices have changed.

Table 1 Changed device names for `block` and `mblock` devices

Old devices names	New device names
/dev/vmedma24d32b	/dev/vmedma24blt
/dev/vmedma32d32b	/dev/vmedma32blt
/dev/vmedma24d32mb	/dev/vmedma24mblt
/dev/vmedma32d32mb	/dev/vmedma32mblt
/dev/fvmedma24d32b	/dev/vmefdma24blt
/dev/fvmedma32d32b	/dev/vmefdma32blt
/dev/fvmedma24d32mblt	/dev/vmefdma24mblt
/dev/fvmedma32d32mblt	/dev/vmefdma32mblt

2.2.3 `ioctl()`

Both drivers, the old-style as well as the new-style driver support the standard UNIX calls. Additionally the new-style driver supports the VME User Interface – VUI. Within this interface there are several VUI function calls which functionally replace the old-style `ioctl()`. Nevertheless, `ioctl()` is supported within the new style driver though it is strongly recommended to use the VUI function calls instead of `ioctl()`.

The `ioctl()` interface has been substantially revised to no longer be CPU-board dependent. Moreover, it is no longer necessary to use `ioctl()` within an application because the VUI can be used instead, providing a hardware-independent VMEbus user interface.
Therefore:

- Applications which are based on the old-style driver and which do not use `ioctl()` will run on the new-style driver as well. For blt and mblt transfers only the device names has to be changed.
- Applications which use `ioctl()` will

204936 9 – 0 January 2000

– either – as in the past – have to adapt the calls to the new `ioctl()`

– or – and this is the solution strongly recommended by Force Computers (especially when developing new applications) – adapt them to VUI functions.

Nevertheless, a list of all supported `ioctl()` calls follows.
For the new-style driver this set of `ioctl()` requests covers all supported Force Computers CPU boards. The VME nexus driver decides whether the `ioctl()` request makes sense or not. If it does not work on the CPU board under consideration, an error code is returned. But the set of `ioctl()` requests does not change. Thereby, only one driver package covers all supported Force Computers CPU boards.
For a detailed description, see the equivalent VUI function and the `ioctl()` description of the respective leaf driver. For an example of how to use `ioctl()` have a look at the VUI source code in `/opt/FRCvme/vui`.

Table 2 **`ioctl()` support by the new-style driver**

Leaf driver	Supported `ioctl()`

`vmeplus`	`VME_RMW`
	`VME_TRANSFER_MODE_SET, …_GET`
	`VME_INTR_ENA, …_DIS`

`vmedma`	`VME_DMA_GET_STATUS` `VME_DMA_INFO`

`vmefdma`	`VME_FDMA_MAP, …_UNMAP`
	`VME_DMA_GET_STATUS`
	`VME_DMA_INFO`

Table 2 **`ioctl()` support by the new-style driver (cont.)**

Leaf driver	Supported `ioctl()`
vmedvma	VME_SLAVE_MAP, …_UNMAP VME_SLAVE_SET
vmectl	VME_REG_READ, …_WRITE VME_REG_BASE_SET, …_GET VME_ARB_MODE_SET, …_GET VME_BRL_SET, …_GET VME_BRM_SET, …_GET VME_BREL_SET, …_GET VME_INTR_GENERATE VME_MBOX_SET, …_REMOVE VME_MBOX_INFO VME_MBOX_ENABLE, …_DISABLE VME_MBOX_WAIT VME_BOARD VME_INTERFACE VME_ABORT_INTR VME_ACFAIL_INTR VME_SYSFAIL_INTR VME_SYSFAIL_ASSERT, …_DEASSERT VME_RESET

2049369 – 0 January 2000

The following table shows the old-style `ioctl()` requests and the respective new-style `ioctl()` requests. Note that the new-style driver supports more `ioctl()` requests than listed. For more information about the `ioctl()` requests of the new-style driver see the respective `ioctl()` description of the related leaf driver and the related VUI call.

Table 3 Relating old-style to new-style driver `ioctl()` requests

`ioctl()` old-style	**`ioctl()` new-style**
`VME_MAP_SLAVE`	`VME_SLAVE_MAP`
`VME_UNMAP_SLAVE`	`VME_SLAVE_UNMAP`
`VME_SLAVE_SET_MAP`	`VME_SLAVE_SET`
`VME_SET_SLAVE_WIN` `VME_SET_SLAVEWIN`	no longer necessary. Instead, the new-style driver sets up the slave window automatically.
`VME_GET_SLAVE_WIN` `VME_GET_SLAVEWIN`	no longer necessary
`VME_GET_SLAVEWPERR`	no longer available. To control write posting errors, the new-style driver can be configured via the appropriate flag entry in `/etc/system` (see section 2 “Installation and Configuration Guide” on page 11) and via the `VME_TRANSFER_MODE_SET` `ioctl()` request.

`VME_SET_REG`
`VME_SET_VSIA16BASE`
`VME_ENA_VSIA16`
	`VME_REG_BASE_SET`

`VME_GET_REG`
`VME_GET_VSIA16BASE`
`VME_DIS_VSIA16`
	`VME_REG_BASE_GET`

`VME_SET_VME_WIN`	no longer necessary. Instead, the new-style driver sets up the master window automatically.
`VME_GET_VME_WIN`	no longer necessary.
`SET_ABORT_PID`	`VME_ABORT_INTR`
`VME_ASSERT_SYSFAIL`	`VME_SYSFAIL_ASSERT`
`VME_NEGATE_SYSFAIL`	`VME_SYSFAIL_DEASSERT`
`VME_SYSFAIL_STAT`	not available.
`VME_DISWP`	`VME_TRANSFER_MODE_SET`

Table 3 Relating old-style to new-style driver `ioctl()` requests (cont.)

`ioctl()` old-style	`ioctl()` new-style
VME_ENAWP	VME_TRANSFER_MODE_SET
VME_INSTALL_MBOX	VME_MBOX_SET VME_MBOX_ENABLE
VME_REMOVE_MBOX	VME_MBOX_DISABLE VME_MBOX_REMOVE
VMEMBOX_WAIT VME_MBOXWAIT	VME_MBOX_WAIT
VME_LED_SET	not available.
VME_LED_GET	not available.

2.3 Examples

The Solaris VMEbus Driver package includes examples for VME devices. All examples are located in `/opt/FRCvme/examples.` The directory includes a `Makefile` to compile the examples and a `README` file providing detailed information for each example – including information on further help provided.

204936 9 – 0 January 2000

3 Installation and Configuration Guide

This section describes how to install the Solaris VMEbus Driver package `FRCvme` for use with the Force Computers SPARC-based CPU boards running Solaris-x. Other packages of this product are installed analogous.

Overview and installation directories

The name of the package is `FRCvme`. The standard installation path is `/opt`. During installation the directory `/opt/FRCvme` is generated and all drivers and driver configuration files are copied to `/kernel/drv` and `/platform/`*`arch`*`/kernel/drv` and bound into the kernel. Furthermore, the necessary header files and libraries (VUI-Library) are copied to the directories `/usr/include/sys` and `/usr/lib`. After a successful installation, the system has to be rebooted so that the VME drivers are loaded.

Check whether already installed

Use `pkginfo`(1) to find out whether the driver is already installed:

```
# pkginfo -l FRCvme
```

`pkginfo` issues an error message if the `FRCvme` driver is not installed. If an older version of the `FRCvme` package is already installed, it has to be removed first and the system has to be rebooted before installing the new version of the `FRCvme` package:

```
# pkgrm FRCvme
# shutdown -g0 -i6
```

Installing the package

Use `pkgadd`(1M) to install the driver. If you do not want to use the default installation, see also “Optional installation parameters” on page 16. If you want to install the package for a diskless client, see also “Diskless client” on page 16.
To install the driver the user must be `root`. The Solaris VMEbus Driver is delivered on tape or CD-ROM.

Tape

```
# pkgadd -d /dev/rmt/0
```

CD-ROM

If you use a CD-ROM for installation, the following 2 cases can occur:

- The volume manager is running. In this case, the CD-ROM is mounted automatically to the directory `/cdrom`.
- The volume manager is not running. In this case, you have to mount the CD-ROM first. As a mount point an already created empty directory can be used. For example, for a CD-ROM device with SCSI ID 6 (which is quite common) and mount point `/cdrom` enter:

```
# mount -F HSFS -r /dev/dsk/c0t6d0s0 /cdrom
```

CD-ROM with volume manager:

– not running

```
# mount -f hsfs -r /dev/dsk/<scsi-device> <mount-point>
# pkgadd -d <mount-point>/pkg
```

– running

```
# pkgadd -d /cdrom/pkg
```

Diskless client

If you want to install the package for a diskless client, you can either install it on the client's server or on the client itself.

– on the client's server

```
# pkgadd -d /dev/rmt/0 -R /export/root/<clientname>
```

– on the client itself

Before installation on a diskless client, check the `share` options on the server host and the `mount` options on the client host (server: `/etc/dfs/dfstab`; client: `/etc/vfstab`). Usually, the filesystems are exported as read-only filesystems, but the client's root must be exported to allow superuser read/write accesses. For more information see the Solaris user documentation.
It may be the case that some warnings are printed if the package is installed on a diskless client (if `/usr` is mounted read-only). These warnings can be ignored safely (see next note).

Optional installation parameters

It is possible to install only part of the files provided by the Solaris VMEbus Driver package.

Example:

It may be required to install the Solaris VMEbus Driver package without loading and attaching the VMEbus drivers themselves (for example, on a development system without VMEbus interface). This can be accomplished by not installing the files which are planned to be stored in `/kernel/drv` and `/platform/`*`arch`*`/kernel/drv`

2049369 – 0 January 2000

The following steps describe how this is actually done:

1. Choose the files to be installed by choosing a combination of the 3 classes the Solaris VMEbus Driver is split up into:
   - `base` denotes all files in `/opt/FRCvme` (this class is mandatory and must not be omitted),
   - `targ` denotes all files in `/kernel/drv` and `/platform/`*`arch`*`/kernel/drv` (i.e. the VMEbus drivers to be attached),
   - `sysfile` denotes all files in `/usr/include` and `/usr/lib`.
2. Create a file `/tmp/FRCvme.classes` which only includes a single line naming all classes which you want to install, e.g.:

   ```
   CLASSES=”base sysfile”
   ```

   This sample response file lists the classes `base` and `sysfile` but omits `targ` so that the drivers will not be installed.

Note: **Omitting the `targ` class will also suppress the verification step that ensures that the driver package is installed on a supported CPU board.**

Note: **It may be useful (but it is not a requirement) to omit the class `sysfile` when installing on diskless clients where `/usr` is mounted read-only. Otherwise, `pkgadd` displays some warnings, which can be safely ignored.**

Non-interactive installation

To install the driver package without being prompted for input it is required to create a *response* file and pass it as argument to the `pkgadd` command. For further information refer to the documentation of `pkgask` and `pkgadd`.

3.1 Configuration

The default configuration supports all features of the Solaris VMEbus Driver package. Therefore:

- The following section is intended for advanced users, only.
- The information provided in this section is only necessary to configure the installed Solaris VMEbus Driver package when customizing one of the delivered drivers or integrating a customized or a third party leaf driver.

- For all other users the installed Solaris VMEbus Driver package is ready for use right after the installation has been finished.

Delivered drivers

This driver package consists of

- 1 VMEbus nexus driver named `VME`, and
- 5 VMEbus leaf drivers named `vmeplus`, `vmectl`, `vmedma`, `vmefdma`, and `vmedvma`.

For each driver there is a configuration file named *`driver`*`.conf` in `/kernel/drv` or `/platform/`*`arch`*`/kernel/drv`, for example, `VME.conf` or `vmeplus.conf`. A standard configuration file for each driver is included in the package. The standard configuration file describes the name and the class of the driver. If necessary, it also describes VMEbus ranges and interrupts.

Example:

The VMEbus access properties for the `vmeplus` driver are configured in `vmeplus.conf`.

The standard configuration files have been designed to implement a fully featured driver. If you want to change one of the configuration files, e.g., to include interrupts, see the respective man page and section 4 “Application Programmer’s Guide” on page 27 for a detailed description of each leaf driver’s configuration.

The configuration file of the VMEbus nexus driver (VME.conf) may be empty. Nevertheless, the VMEbus nexus driver supports properties for VME master ranges and VME slave ranges. If only the leaf drivers of the Solaris VMEbus Driver package are used together with the VMEbus nexus driver, the configuration file can remain as is. If, however, the VMEbus nexus driver is used with other leaf drivers, see section 5 “Device Driver Developer’s Guide” on page 99 for detailed information on the configuration of the VMEbus nexus driver.

Tuning drivers

The remaining part of this section describes variables which are important to application programmers using leaf drivers.
Most variables specify a configuration detail of the VMEbus nexus driver and thereby potentially affect all VMEbus leaf drivers (see “Access hierarchy” on page 6). If a variable also refers to other drivers, it is explicitly stated in the description of the variable given below (see for example `vme_slave_diswp_flag` and `vme_master_default)`.

To set a variable *`variable`*, which refers to the driver *`driver`*, to the value *`value`*, insert the following line in `/etc/system` (see also the `system(4)` man page):

```
set driver:variable=value
```

2049369 – 0 January 2000

Example:

Setting `vme_diswp_flag` and `vme_master_default`:

```
set VME:vme_diswp_flag=1
set VME:vme_master_default=0x08000000
set vmeplus:vme_master_default=0x08000000
```

`vme_diswp_flag` controls the status of global write posting for master access. If the flag is cleared or not available, all drivers handle write posting for master accesses as defined elsewhere.

= `1` Write posting is disabled for every driver for master windows. Enabling write posting via other flags or VUI functions is impossible.

`vme_slave_diswp_flag` controls the status of global write posting for slave window access. It affects `vmedvma`. If the flag is cleared or not available, all drivers handle write posting for slave windows as defined elsewhere.

= `1` Write posting is disabled for every driver for slave windows. Enabling write posting via other flags or VUI functions is impossible.

`vme_master_default` sets the transfer mode of master windows. For valid values and a general discussion on VME AM code generation see

- `vme_types.h` header file,
- section 4.1.7 “vui_transfer_mode_set(), vui_transfer_mode_get()” on page 40,
- and section 6 “VME Bus Properties” on page 159.

`vme_master_default` can be used

- for the VME nexus driver (`VME`), thereby affecting all VME leaf drivers with the exception of `vmeplus`. Sample `/etc/system` entry for enabling write posting:

  ```
  set VME:vme_master_default=0x01000000
  ```

- for the `vmeplus` leaf driver, thereby only affecting `vmeplus`. This default setting may get overridden by software. Sample `/etc/system` entry for enabling write posting:

  ```
  set vmeplus:vme_master_default=0x01000000
  ```

`vme_mwperr_action` defines how to react on a LocalBus-to-VME write-posted errors which cannot be back tracked to the originating process or driver. For example, this happens when write posting is enabled on the FGA-5000 and a VMEbus access error occurs.
The following value list will be extended in future:

= `0` Print warning only (default).

= `1` Ignore LocalBus-to-VME posted write errors.

= `2` Panic on LocalBus-to-VME posted write errors.

`vme_swperr_action` defines how to react on a VME-to-LocalBus write-posted error (VME-to-SBus errors cannot be back tracked to a process or driver):

= `0` Print warning only (default).

= `1` Ignore VME-to-LocalBus posted write errors.

= `2` Panic on VME-to-LocalBus posted write errors.

`vme_uwerr_action` defines how to react on a non-posted user write error (for example, when the VMEbus is accessed via `mmap()`).

Note: **The handling capabilities for VMEbus write errors differ significantly depending on the type of hardware architecture used. The default reaction is therefore very conservative. Refer to the *Release Notes* for information on whether the behavior can be modified for the hardware under consideration.**

= `0` When a user access occurs, print a warning and send the SIGBUS (bus error) signal.

= `1` Ignore VME-to-LocalBus non-posted user-write errors.

= `2` Send the SIGBUS (bus error) signal.

= `3` Panic.

= `4` Print a warning (default).

`vme_kwerr_action` defines how to react on a non-posted kernel write error.

Note: **The handling capabilities for VMEbus write errors differ significantly depending on the type of hardware architecture used. The default reaction is therefore very conservative. Refer to the *Release Notes* for information on whether the behavior can be modified for the hardware under consideration.**

= `0` Print a warning and send a SIGBUS (bus error) signal.

= `1` Ignore VME-to-LocalBus non-posted kernel write errors.

= `2` Send the SIGBUS (bus error) signal.

= `3` Panic.

= `4` Print a warning (default).

`vme_fault_hndl_off` controls whether the Solaris fault handling routines will not be changed. This causes unpredictable results when VMEbus errors occur.

= `0` The Solaris fault handling routines may be changed (default).

= `1` The Solaris fault handling routines will not be changed.

`vme_krerr_action` defines how to react on kernel read faults:

2049369 – 0 January 2000

= `0` `copyout()` reports an error and other accesses cause a bus error signal to the process (default).

= `1` Same as `0`, additionally a warning is printed.

`vme_rerr_stall` defines how to react on read errors. When a VMEbus memory location is read and a bus error occurs, the processor issues a synchronous (precise) trap. This activates the error recovery function of the VMEbus nexus driver, which sends a SIGBUS signal to the offending process. Depending on `vme_rerr_stall`, 2 continuations are possible:

= `0` This setting is only available on sun4m hardware architectures: Skip the machine instruction and resume the process with the next instruction. This results in the original read instruction returning data which seem to be valid, although the read transaction has actually failed and the data content is indetermined. When using this setting, the programmer should also catch SIGBUS to correctly deal with data returned by the read operation although the transfer failed.

= `1` Restart the machine instruction which caused the precise trap (default). This usually results in the VMEbus accessing the address which caused the bus error over and over again, i.e. the error recovery function does not stop sending SIGBUS signals to the process.

`vme_event_warn` defines how to react on ACFAIL, SYSFAIL, and ABORT:

= `0` Do not print any warning (default).

= `1` Print a warning, if the corresponding event is not handled.

3.2 Basic Test of the Driver

After successful installation the package's program examples can be used to access a slave board on the VMEbus. Ensure to have rebooted the system after package installation.

Sample situation for the screen output shown in this section:

The following examples assume that there is a memory board at VMEbus address 6000.0000_{16}, accessible in the A32 address range, which accepts 32-bit single transfers.

Filling memory with 0

1. Use the `vmedma` driver to fill the first megabyte of the memory board with `0`.

```
# cd /opt/FRCvme/examples
# vmecp -t -a 60000000 -s 100000 /dev/zero /dev/vmedma32d32
1048576 bytes in 0.08 real seconds = 12.31 MB/sec
```

Write access

2. Use the `vmeplus` driver to write some data to the memory board. `vme_dump` takes data from standard input and writes it to the specified VMEbus address until CTRL-D is entered.

```
# vme_dump -w /dev/vme32d32 60000000
This is a test
^D
```

Read access

3. Use the `vmeplus` driver to read the data we just wrote to the memory board. In the example below, 20_{16} (= 32_{10}) Byte are read.

```
# vme_dump -r /dev/vme32d32 60000000 20
This is a test
```

Read and store in file

4. Use the `vmedma` driver to read 1 megabyte and store it in a file.

```
# vmecp -t -a 60000000 -s 100000 /dev/vmedma32d32 /usr/tmp/data
1048576 bytes in 0.16 real seconds = 6.39 MB/sec
# cat /usr/tmp/data
This is a test
# rm /usr/tmp/data
```

2048369 – 0 January 2000

Troubleshooting

If a VMEbus bus error occurs, `vme_dump` will terminate with a core dump, `vmecp` with an I/O error. If this happens, make sure that

- the respective CPU board is present and accessible at the VMEbus address specified, and that it does not conflict with other bus participants.
- the respective CPU board accepts extended supervisory data accesses (AM code $0D_{16}$).

For a detailed troubleshooting discussion, see next section.

3.3 Troubleshooting

If the installation aborts, the package is considered to be only partially installed. An appropriate message will be displayed before ending the installation. Have a second try on the package installation after checking the following:

Older version already installed

Check whether an older version has not been uninstalled before installing the new version of the Solaris VMEbus Driver. Also note that the system has to be rebooted after installing or uninstalling a nexus driver as is true for the Solaris VMEbus Driver package.

Irregular boot termination

If the boot process terminates before the VMEbus driver is loaded (boot with the option `-v` to see every boot message), this may be caused by

- some hardware defect: see respective manual to locate the defect),
- incorrect OpenBoot boot settings: consult the respective CPU board's manual to figure out the default setting and use the CPU board's default setting for a retry,
- or the standard Solaris `mcp` driver trying to access the VMEbus hardware: Solaris 2.x includes a driver of the `vme` class (see `/kernel/drv/mcp`). Unfortunately, `mcp` assumes that it found an `mcp` board if accessing certain VMEbus addresses is possible and does not double check this assumption, e.g by accessing a control register. This can cause the VMEbus system to hang or to panic if there is accidentally another board installed at one of the addresses accessed by `mcp`. To avoid this, the `mcp` driver should be removed.

Hang up after VMEbus driver message

If the system hangs without any further messages after the VMEbus driver message is displayed, the system is trying to access the VMEbus but the access is not successful.

– system contr.

Check whether there is no system controller (arbiter) in the VMEbus rack or the CPU board's system controller function has been disabled. Check the respective switch setting for the VMEbus slot-1 configuration (e.g. check the VMEbus slot-1 auto-configuration and the VMEbus slot-1 manual mode switch, if implemented on the CPU board). Ensure

- that there is a system controller present,
- that there is only one system controller in the VMEbus rack,
- and that it is plugged into slot 1 of the VMEbus rack.

– address ranges

When there are several CPU boards installed in the rack,

- check the VMEbus addresses of all installed CPU boards. The VMEbus address ranges of the CPU boards must be non-overlapping.
- make sure that the master CPU board does not try to decode its own VMEbus addresses. For example, this happens if the `slavewin` property is enabled (see section 5.1 "VME Nexus Driver Configuration" on page 99) and some device driver is configured to use this address range.

Missing drivers

Unlike in former Solaris versions, no drivers – be it nexus or be it leaf drivers – will be loaded permanently by default. Instead, Solaris 2.x dynamically loads every driver when it is needed. To check which drivers are loaded, use the `modinfo` command. Due to dynamic loading, it might well be that no driver is loaded after a reboot. Only when booting with the reconfigure option `-r`, any installed driver will be registered.

Note: **Some VME leaf drivers may not be loaded on hardware that lacks the corresponding hardware features. For example, the `vmedma` and `vmefdma` drivers cannot be installed on S4 based boards, since the S4 SBus-to-VME bridge has no DMA controller.**

2049369 – 0 January 2000

3.4 Limitations

The user interface of this software package stays the same regardless of the type of the CPU board this package is installed on. However, the following limitations apply:

- A function call may be parameterized by values that are depending on the type of CPU board used: for example, reading a register depends on the register's address which in general is specific to the CPU board and the VMEbus interface chip being used on the CPU board. Therefore, a header file is available for each interface chip containing the `#define` statements for each available register address.
- A function call does not necessarily have an effect: If the CPU board does not support a feature required to execute a given function call, the function can be called but it will return without having any effect. For example, there are CPU boards
  - providing only 1 master window, whereas others provide up to 16.
  - being unable to generate VMEbus interrupts.

  For information on such limitations, see the *Solaris VMEbus Driver Release Notes* of the release under consideration and the CPU board's *Technical Reference Manual*.
- The following limitation results from the Sun 4m architecture: The `vmefdma` and `vmedvma` drivers need kernel memory for their buffers. For Sun 4m architectures, the kernel memory is limited to 100 MByte. However, not all of the kernel memory is available at run-time. The longer the system is up running, the more fragmented the kernel memory becomes. Further more, the used memory size is limited by the IOMMU. The IOMMU address space is maximally 64 MByte, and just as the kernel memory it becomes the more fragmented the longer the system has been up running.

204936 9 – 0 January 2000

4 Application Programmer's Guide

Application programmers need to know the interfaces between the leaf driver and the application. The Force Computers leaf drivers provide 2 such interfaces:

1. the standard UNIX system calls: For information on them, see the man pages and the respective Solaris manual on system calls.
2. the VUI – the VME user application interface which is an additional layer of abstraction between a user application and the device drivers for VME related functions.

Both interfaces are described in this section for every leaf driver included in the Solaris VMEbus Driver package.

Using VUI functions

To use VUI functions, the application must be linked with the VUI library `libvui.a` in `/opt/FRCvme/usr/lib/` or `/usr/lib/`:

```
# cc -o app app.c -L/opt/FRCvme/usr/lib -lvui
```

or:

```
# cc -o app app.c -L/usr/lib -lvui
```

Example

For examples on how to access the VMEbus see the sample source code in `/opt/FRC/examples`.

4.1 vmeplus

vmeplus replaces Sun's vmemem driver. It includes all the code necessary to map and access VMEbus memory via programmed I/O. Additionally, it provides handlers for VMEbus interrupts which forward VMEbus interupts as signals to processes. It is a standard character device driver.

Devices

By default, the driver provides access to a number of device nodes in /dev which are named as follows:
/dev/vme<*space*><*data*>

Where <space> is

- 16 for accessing data in A16 address spaces,
- 24 for accessing data in A24 address spaces,
- 32 for accessing data in A32 address spaces,
- crcsr for accessing data in the CR/CSR address space, and
- pam1 or pam2 for accessing data in 1 of the 2 user-defined address spaces. The generated AM code depends on the configuration of the VME Nexus driver (see section 5.1 "VME Nexus Driver Configuration" on page 99).

<data> denotes the way the data is transferred on the VMEbus:

- d8 for 1-byte single cycles,
- d16 for 2-byte single cycles (including 1-byte cycles),
- d32 for 4-byte (lword) single cycles (including 1- and 2-byte cycles),
- blt for BLT burst cycles (including all single cycles),
- mblt for MBLT (D64) burst cycles (including BLT burst and all single cycles), and
- te for 2-edge burst cycles (including all other burst and single cycles).

As can be seen, the <*data*> identifier denotes the maximum data capability that will be generated on the VMEbus. However, the actual transfer depends on the load/store operation(s) done by the processor. For example, accessing an 1-byte entity via the /dev/vme32d16 device will result in 1-byte single cycles on the VMEbus, but a 4-byte access will be split into two 2-byte transfers on the VMEbus.

Note: **The availability of these devices depends on the hardware capabilities of the VMEbus bridge used. Refer to the *Release Notes* for details.**

2049369 -- 0 January 2000

Routines

To access the driver and its devices the following routines are supported:

- UNIX system calls: `open()`, `close()`, `read()`, `write()`, `mmap()`, `munmap()`, `ioctl()`.
- VUI calls: `vui_intr_ena()`, `~_dis()`, `vui_rmw()`, `vui_transfer_mode_set()`, `~_get()`.

Configuration

`vmeplus` overrides any default settings imposed by the VME nexus driver with one exception: `VME:vme_diswp_flag` (see section 3.1 "Configuration" on page 17).

The write posting property (off by default), which is related to VMEbus accesses done by `vmeplus`, is not controlled by selecting the appropriate minor node. It primarily depends on the hardware capabilities whether this property can be controlled at all and if so, to which state they are set. Refer to the corresponding section in the *Release Notes* for details.

To change the default behavior,

- modify the `vmeplus:vme_master_default` parameter in `/etc/system` (see `system(4)`). The parameter's value is a bitmask containing bus property bits (see section 6 "VME Bus Properties" on page 159).
- or use `vui_transfer_mode_set()` (see page 40). This overrides the `vmeplus:vme_master_default` setting.

Example:

To enable write posting for VMEbus accesses by `vmeplus`, insert the following line in /etc/system:

```
set vmeplus:vme_master_default=0x1000000
```

Configuration file

`/kernel/drv/vmeplus.conf` is the vmeplus configuration file. The file ends with a semicolon.

Sample configuration file:

```
interrupts=4,0x4c,5,0x50
reg=0x2d,0,0x10000,0x3d,0,0xff0000,
  0x0d,0,0xff000000,0x6d,0,0x10000,
  0x7d,0,0xff0000,0x4d,0,0xff000000,
  0x2f,0,0x1000000,0x6f,0,0x1000000,
  0x10,0,0xff000000,0x11,0,0xff000000,
  0x50,0,0xff000000,0x51,0,0xff000000
name="vmeplus" class="vme";
```

– name — Ensure that the name="vmeplus" statement is always included in the configuration file as described in the sample configuration file. The driver name in this case is always vmeplus.

– class — Ensure that the class="vme" statement is always included in the configuration file as described in the sample configuration file. Thereby, the actual driver name for the VMEbus interface which may vary across hardware platforms is hidden from the leaf drivers and the parent is specified in terms of its interface type, instead: Via the class="vme" statement, they relate themselves to the nexus driver of class vme.

– interrupts — The interrupts property is a comma separated list of pairs:

- the first entry in a pair being the VMEbus IRQ level
- and the second being a VMEbus IRQ vector number.

The first interrupt property (IRQ#4 vector 0x4c) has the property number 0, the second pair of values has the property number 1 and so on. This is important for the vui_intr_ena() and vui_intr_dis() functions (see section 4.1.5 "vui_intr_ena(), vui_intr_dis()" on page 37).

Sharing the same interrupt vector among several interrupt levels is possible.

– reg — The reg property is a comma separated list of triples which define bus type, the start address and end address of accessible VMEbus areas. Refer to section 5.2.2 "VMEbus Mappings" on page 105 for details on the format of VMEbus reg properties.
The order of the reg-triples is unimportant. The vmeplus driver will create device nodes in /dev based on the number of triples present, and the names are defined based on the information found in the bus type fields.
The device names generally do not reflect whether non-privileged, supervisory, program or data accesses are generated on the VMEbus.

Absolute data widths — As already mentioned, the data width generated by programmed I/O may alter, e.g. a 1-byte access may be done even when using the vmexxd32 device. This is especially the case when using the read()/write() system calls.
By setting one or both of the properties absolute-width-read and absolute-width-write into vmeplus.conf to 1, the driver will alter the behavior of the read() and write() system calls respectively. The driver will make sure that the size of VME single cycles are made with the same data size as defined by the name of the device used to perform the accesses. Both source and destination addresses must be properly aligned.

2049369 – 0 January 2000

Note: **Setting the `absolute-width-xxx` properties significantly degrades the data throughput of the `read()/write()` system calls.**

4.1.1 open(), close()

SYNTAX

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(
          const char *path, /* path to device node */
          int oflag);/* Open Flags*/

#include <unistd.h>
int close(int filds);/* File handle of opened device */
```

DESCRIPTION

`open()`
obtains access to a VMEbus device and prepares it for use.

`close()`
closes a VMEbus file descriptor associated with a VMEbus device.

Both operations are similar to standard `open(2)` and `close(2)`. For further information, see the respective man pages.

RETURN VALUES

On successful completion, `open()` returns the file descriptor. Otherwise, `-1` is returned and `errno` is set to indicate the error. `close()` always returns `0`. See the `open(2)` and `close(2)` man pages.

ERRORS

See also man pages of `open(2)` and `close(2)`.

ENXIO
The minor node of the device is not supported. E.g. `vmecrcsr16` is not supported for FGA-5000 based CPU boards.

4.1.2 read(), write()

SYNTAX

```
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>
size_t read(
          int filds, /* File handle of opened dev */
          void *buf,/* buffer to receive data */
          size_t nbyte);/* number of bytes to transfer */

#include <unistd.h>
size_t write(
          int filds,/* File handle of opened dev */
          const void *buf,/* buffer containing data */
          size_t nbyte);/* number of bytes to transfer */
```

DESCRIPTION

`read()`
copies a block of data from the VMEbus address space to a user process buffer.

`write()`
copies a block of data from a user process buffer to the VMEbus address space.

The `read()` and `write()` function calls implement reading of or writing to a previously opened VME device by using programmed I/O. For higher performance use the DMA or fast DMA driver (`/dev/vmedma`*xx*`d`*yy* or `/dev/vmefdma`*xx*`d`*yy*) which provide a similar interface.
The VMEbus address to be accessed is defined by the file pointer, which in turn can be set by `lseek()` and `llseek()`.

Both operations are similar to standard `read(2)` and `write(2)`. For further information, see the respective man pages.

RETURN VALUES

On successful completion, the number of bytes transferred is returned. Otherwise, `-1` is returned and `errno` is set to indicate the error (see the `read(2)` and `write(2)` man pages).

ERRORS

See also the `read(2)` and `write(2)` man pages.

EAGAIN
At the time of execution the action cannot be done, e.g. because no resources are available.

EINTR
The process has been interrupted by a signal while waiting for resources to become available.

EINVAL
Invalid argument.

D2049369 -- 0 January 2000

ENOTSUP
Action not supported.

EFAULT
The system call has been terminated due to a bus error on the VMEbus.

Note: **The `write()` system call usually cannot detect whether a write access has been terminated by a VME bus error, i.e. it will not return an error code. At least in case of VME write posting being disabled, error detection can be enabled by setting the `absolute-width-write` property in the driver configuration file (refer to "Absolute data widths" on page 30 for details).**

EXAMPLE

```
{
  ...
  int vmedev;
  int rc, i;
  char buf[100];

  if ((vmedev=open("/dev/vme32d32", O_RDWR)) == -1)
  {
      perror("open"); … /* ERROR */
  }
  (void)printf("filling the write buffer\n");
  for ( i = 0;  i < 100; i++ )
  {
         buf[i]=i;
  }
  (void)printf("seeking to vme address 0x60000000\n");
  if (lseek(vmedev, 0x60000000, SEEK_SET) == -1)
  {
         perror("lseek"); … /* ERROR */
  }
  (void)printf("writing the buffer to the VME memory\n");
  rc = write ( vmedev, buf, 100 );
  if ( rc < 0)
  {
         perror("write"); … /* ERROR */
  }
  (void)printf("seeking to vme address 0x60000000\n");
  if (lseek(vmedev, 0x60000000, SEEK_SET) == -1)
  {
         perror("lseek"); … /* ERROR */
  }
  rc = read ( vmedev, buf, 100 );
  if (rc <= 0)
  {
      perror("read"); … /* ERROR */
  }
  ...
  (void)close(vmedev);
  ...
}
```

4.1.3 mmap(), munmap()

SYNTAX

```
#include <sys/types.h>
#include <sys/mman.h>
caddr_t mmap(
          caddr_t addr,/* virtual address hint */
          size_t len, /* # bytes to map in */
          int prot, /* protection mode */
          int flags,/* flags for page handling */
          int fildes, /* VME dev file handle */
          off_t off);/* VMEbus address */

#include <sys/types.h>
#include <sys/mman.h>
int munmap(
          caddr_t addr,/* usr addr for mapped VME block */
          size_t len);/* block size mapped in bytes */
```

DESCRIPTION

`mmap()`
allows a VME address range to be mapped into an application's address space.

`munmap()`
removes a previously set up mapping. Partial unmapping is not supported.

Note: **Even though the argument `off` is specified to be a signed value a full 32-bit VMEbus address for A32 devices may be used. The driver will interpret it as a 32-bit unsigned value.**

Note: **It is not possible for a process to set up a mapping whose address range overlaps a range previously mapped for the same device by this process.**

The type of the mapped VMEbus address space is determined by the opened device node. Additional parameters, like enabling write posting, can be set via `vui_transfer_mode_set()` (see page 40).
Mapped VMEbus address ranges and their properties are inherited when a process forks.
Mappings must be set up as shared, i.e. the `flags` parameter has to be set to `MAP_SHARED`.
The requested VMEbus address must be aligned to the hardware page boundary (see `getpagesize(3C)`).

RETURN VALUES

`mmap()`
On successful completion, `mmap()` returns the start address within the application's address space to which the VME device has been

2049369 – 0 January 2000

mapped. Otherwise, it returns `MAP_FAILED` and sets `errno` to indicate the error (see the man page for `mmap(2)`).

`munmap()`
On successful completion, `munmap()` returns `0`. Otherwise, it returns `-1` and sets `errno` to indicate the error (see `munmap(2)`).

ERRORS

See also the man page for `mmap(2)` and `munmap(2)`.

EAGAIN
At the time of execution the action cannot be done, e.g. because no resources are available.

EINTR
The process has been interrupted by a signal while waiting for resources to become available.

EINVAL
Invalid argument.

ENOTSUP
Action not supported.

EXAMPLE

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>

void main()
{
    int vmedev, i;
    char *cptr, *v_vmeaddr;

    /* Open device */
    if ((vmedev = open("/dev/vme32d32", O_RDWR)) == -1) {
          perror("/dev/vme32d32"); exit(1);
    }

    /* Map 240 MBytes of VME memory */
    v_vmeaddr = mmap( 0, 240*1024*1024,PROT_READ|PROT_WRITE,
                   MAP_SHARED, vmedev, 0x60000000 );

    if (MAP_FAILED == v_vmeaddr) {
          perror("mmap"); exit(1);
    }

    /* write zeros to VME using D8 (char) single cycles.  */
    printf("Clearing 240 MB memory at VME 0x60000000."
          "Press CTRL-C to abort\n");
    for (cptr = v_vmeaddr; cptr < v_vmeaddr+240*1024*1024;
         cptr++)
          *cptr = 0;
    exit(0);
}
```

4.1.4 ioctl()

SYNTAX

```
#include <unistd.h>
#include <sys/vme_types.h>
#include <sys/vme.h>
#include <sys/vui.h>
int ioctl(int fildes, int request, /* arg */ ...);
```

DESCRIPTION

`ioctl()`
performs various device-specific control functions on devices. `request` and an optional third argument with varying type are passed to the file designated by `fildes` and are interpreted by the device driver. For further information see man pages of `ioctl(2)`.

Note: **It is strongly recommended to use VUI functions instead of `ioctl()`.**

VARIABLES

`fildes`
file descriptor of an opened VME device

`request`
selects the control function to be performed and depends on the device being addressed. The following requests are defined:

```
VME_RMW
VME_TRANSFER_MODE_SET
VME_TRANSFER_MODE_GET
VME_INTR_ENA
VME_INTR_DIS
```

For a description of how these requests work see the respective VUI function (`vui_rmw()`,...). For examples how to use `ioctl()`, see the source code of the VUI functions.

`arg`
parameter that might be needed by the specified device to perform the requested function. The data type of `arg` depends on the particular control request, but it is either an `int` or a pointer to a device-specific data structure.

RETURN VALUES

On successful completion, the value returned depends on the device control function, but it is always a non-negative integer. Otherwise, `-1` is returned and `errno` is set to indicate the error.

ERRORS

See man pages of `ioctl(2)` and VUI functions

204936 9 – 0 January 2000

4.1.5 vui_intr_ena(), vui_intr_dis()

SYNTAX

```
#include <sys/vme_types.h>
#include <sys/vme.h>
#include <sys/vui.h>
int vui_intr_ena (int dev, ioctl_irq_t *intr)
int vui_intr_dis (int dev, ioctl_irq_t *intr)
```

DESCRIPTION

`vui_intr_ena()`
enables the VMEbus interrupt defined in the structure `intr` and forwards it as signal to the calling process.

`vui_intr_dis()`
disables the interrupt defined in the structure `intr`.

VARIABLES

`dev`
file descriptor of an opened VME device

`*intr`
pointer to interrupt definition struct `ioct_irq_t`

The struct `ioctl_irq_t` is defined in `vme.h`:

```
struct ioctl_irq
{
  int   prop; /* …ena in , …dis in:
           * property from vmeplus.conf file */
  int   sig; /* …ena in: signal, -1 means no signal */
  int   level; /* not relevant for these calls */
  int   vector;/* not relevant for these calls */
};
typedef struct ioctl_irq      ioctl_irq_t;
```

`prop`
defines the index (zero based) of the interrupt property pair. Each interrupt must be defined in the `vmeplus` driver's configuration file (`kernel/drv/vmeplus.conf`) by means of an interrupt property, which describes level and vector. Input for `vui_intr_ena()` and `…_dis()`.

`sig`
sets the signal which shall be sent to the user application when the interrupt occurs. Input for `vui_intr_ena()`.

> Not all signals can be sent to an application. For a list of possible signals, see the `proc_signal(9F)` man page.

`level`
interrupt level, for these calls not relevant and therefore undefined.

`vect`
interrupt vector, for these calls not relevant and therefore undefined.

By default, there can be 7 interrupts at maximum defined in the `vmeplus.conf` file. The interrupt property itself can be set to the user's needs.

RETURN VALUES

On successful completion, `VUI_OK` is returned. Otherwise, `VUI_FAIL` is returned and `errno` is set to indicate the error.

ERRORS

See also man pages of `ioctl(2)`

EACCES
In case of `vui_intr_dis()` the interrupt to be disabled has been enabled by another process.

EAGAIN
In case of `vui_intr_ena()` the interrupt is already enabled.

EINVAL
Invalid argument, e.g. because the `sig` parameter of the request structure denotes a signal that cannot be used, or because the `prop` parameter does not reference an existing entry in `vmeplus.conf`.

ENOTSUP
Action not supported.

EXAMPLE

```
int sighdl( int arg )
{
  intr_count++;
}
...

void test_funct( void )
{
  ...
  int vmedev;
  ioctl_irq_t intr;
  int dummy;

  /* open some vmeplus device */
  if ((vmedev=open("/dev/vme32d32", O_RDWR)) == -1)
  {
      perror("open"); … /* ERROR */
  }

  intr.prop = 0;
  intr.sig = SIGINT
  intr_count = 0;
```

204936 9 – 0 January 2000

```
  /* set the signal handler */
  sigset( SIGINT, sighdl );
  /* activate the interrupt */
  vui_intr_ena(vmedev, &intr);

  /* receive ten interrupts */
  while (intr_count < 10)
  {
          wait( &dummy );
          printf( "Interrupt %d received\n", intr_count );
  }

  vui_intr_dis( vmedev, &intr );
  sigset( SIGINT, SIG_DLF );
  (void)close(vmedev);
  ...
}
```

4.1.6 vui_rmw()

SYNTAX

```
#include <sys/vme_types.h>
#include <sys/vme.h>
#include <sys/vui.h>
int vui_rmw (
          int dev,
          vmeaddr_t vmeaddr,
          ubyte_t *value);
```

DESCRIPTION

`vui_rmw()`
performs a read-modify-write cycle (load and store unsigned byte)

VARIABLES

`dev`
file descriptor of an opened VME device

`vmeaddr`
VME address for the transaction

`value`
value to be written to the VME address.

Note: **The hardware implementation may limit the values that can actually be written. Refer to the *Release Notes* for details on the board under consideration.**

RETURN VALUES

On successful completion, `VUI_OK` is returned. Otherwise, `VUI_FAIL` is returned and `errno` is set to indicate the error.

ERRORS

See also man pages of `ioctl(2)`

EAGAIN
At the time of execution the action cannot be done, e.g. because no resources are available.

EFAULT
A VMEbus error occurred during the transaction.

EINVAL
Invalid argument, or no more ranges are available.

ENOTSUP
Action not supported.

EXAMPLE

```
{
  ...
  int vmedev;
  int value;
  if ((vmedev=open("/dev/vme32d32", O_RDWR)) == -1)
  {
      perror("open"); … /* ERROR */
  }
  value=0xff;
  vui_rmw(vmedev, (vmeaddr_t) 0x60000000, &value);
  ...
  (void)close(vmedev);
  ...
}
```

4.1.7 vui_transfer_mode_set(), vui_transfer_mode_get()

SYNTAX

```
#include <sys/vme_types.h>
#include <sys/vme.h>
#include <sys/vui.h>
int vui_transfer_mode_set (int dev, bt_t tm)
int vui_transfer_mode_get (int dev, bt_t *tm)
```

DESCRIPTION

`vui_transfer_mode_set()`
sets the transfer mode for all following master accesses done via `mmap()`, `read()` / `write()`, or `vui_rmw()`. Only those bus properties are relevant which are masked by VME_BT_TMASK.

`vui_transfer_mode_set()` allows to change the driver's default behavior. Alternatively, `vmeplus:vme_master_default` can be set appropriately in `/etc/system` (see section 3.1 "Configuration" on page 17).

`vui_transfer_mode_get()`
returns the actual transfer mode in `tm`.

VARIABLES Variables for `vui_transfer_mode_set()`:

`dev`
file descriptor of an opened VME device

`tm`
flags defining the transfer mode(s) to enable or disable. Only the flags masked by `VME_BT_TMASK` are relevant (see `sys/vme_types.h` and section 6 "VME Bus Properties" on page 159): `VME_BT_WP`, `~_PF`, `~_UNALIGN`, `~_PRIAUTO`, and `~_PROGAUTO`.

To enable or disable the transfer mode(s) described by the asserted bit(s), the `tm` value can be the logical OR of one of the following values:

`VUI_SET`	Set this flag to enable the specified transfer modes.
`VUI_CLEAR`	Set this flag to disable the specified transfer modes.

Variables for `vui_transfer_mode_get()`:
`dev`
file descriptor of an opened VME device

`*tm`
pointer for the current bus properties. Within this pointer the current bus properties are returned. Possible bus properties are defined within the header file `vme_types.h`. (see `sys/vme_types.h` and section 6 "VME Bus Properties" on page 159).

Note: **It is not possible to set the transfer modes for individual device nodes or processes. At the moment when the transfer mode is set up, it is valid for the whole driver instance. For the `vmeplus` driver this means the following: if the transfer mode is set up for example for `/dev/vme32d32`, it is valid for all `/dev/vme`*`xx`*`d`*`yy`* devices and for all other processes using these devices. However, existing mappings will not be affected.**

RETURN VALUES
On successful completion, `VUI_OK` is returned. Otherwise, `VUI_FAIL` is returned and `errno` is set to indicate the error.

ERRORS
See also man pages of `ioctl(2)`

EINVAL
Invalid request or argument.

EXAMPLE

```
{
  ...
  int vmedev;
  bt_t tm;

  if ((vmedev=open("/dev/vme32d32", O_RDWR)) == -1)
```

```
    {
        perror("open"); … /* ERROR */
    }

    /* disable write posting for all vmeplus devices */
    vui_transfer_mode_set(vmedev, VUI_CLEAR | VME_BT_WP);
    ...
    /* read out the actual transfer mode */
    vui_transfer_mode_get(vmedev, &tm);

    (void)close(vmedev);
    ...
}
```

2049369 – 0 January 2000

4.2 vmedma

The vmedma driver utilizes the on-board DMAC (direct memory access controller). It contains all the code necessary to initiate a transfer from user space to VMEbus memory, and vice versa.

Devices

By default, the driver provides access to a number of device nodes in /dev which are named as follows:
/dev/vmedma<*space*><*data*>

Where <space> may be

- 16 for accessing data in A16 address spaces,
- 24 for accessing data in A24 address spaces,
- 32 for accessing data in A32 address spaces,
- crcsr for accessing data in the CR/CSR address space, and

<data> denotes the way the data is transferred and may be

- d8 for 1-byte single cycles,
- d16 for 2-byte single cycles (including 1-byte cycles),
- d32 for 4-byte (lword) single cycles (including 1- and 2-byte transfers),
- blt for BLT burst cycles (including all single cycles),
- mblt for MBLT (D64) burst cycles (including BLT burst and single cycles), and
- te for 2-edge burst cycles (including all other burst and single cycles).

Routines

To access the driver the following routines are supported:

- UNIX system calls: open(), close(), read(), write(), ioctl().
- VUI calls: vui_dma_malloc().

Configuration file

/kernel/drv/vmedma.conf is the vmedma configuration file. It contains a reg property which defines the vmedmaxxx device nodes created in /dev. For a on the format of the reg properties refer to section 5.2.2 "VMEbus Mappings" on page 105.
The reg property may be modified for accessing VMEbus address spaces not present in the default configuration.

4.2.1 open(), close()

SYNTAX

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(
          const char *path,/* path to device node */
          int oflag);/* Open Flags*/

#include <unistd.h>
int close(int filds);/* File handle of opened device */
```

DESCRIPTION

`open()`
obtains access to the VMEbus device and prepares it for use.

`close()`
closes a VMEbus file descriptor associated with a VMEbus device.

Both operations are similar to standard `open(2)` and `close(2)`. For further information, see the respective man pages.

RETURN VALUES

On successful completion, the file descriptor is returned. Otherwise, `-1` is returned and `errno` is set to indicate the error (see the `open(2)` and `close(2)` man pages).

ERRORS

See also the `open(2)` and `close(2)` man pages

ENXIO
The minor node of the device is not supported. E.g. `vmedma32te` is not supported for FGA5000-based CPU boards.

4.2.2 read(), write()

SYNTAX

```
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>
size_t read(
          int filds, /* File handle of opened dev */
          void *buf,/* buffer to receive data */
          size_t nbyte);/* number of bytes to transfer */

#include <unistd.h>
size_t write(
          int filds,/* File handle of opened dev */
          const void *buf,/* buffer containing data */
          size_t nbyte);/* number of bytes to transfer */
```

DESCRIPTION

`read()`
copies a block of data from the VMEbus address space to a user process buffer.

2049369 -- 0 January 2000

`write()`
copies a block of data from a user process buffer to the VMEbus address space.

The `read()` and `write()` function calls implement reading of or writing to a previously opened VME device. Via read and write system calls the entire 32-bit VME space is accessible. VME D32, D16, BLT, MBLT, and 2eVME accesses are supported (2eVME as of Solaris VMEbus Driver release 2.1).
The value of the file pointer can be set using `lseek()` and `llseek()`. With the help of these 2 function calls, the starting VME address for read or write access can be specified.

Both operations are similar to standard `read(2)` and `write(2)`. For further information, see the respective man pages.

Note: **Some hardware needs properly aligned buffer and/or VMEbus addresses (refer to the *Release Notes* if this is true for the CPU board you use). To allocate the DMA buffer, it is recommended to use the VUI function `vui_dma_malloc()` which allocates properly aligned memory (see section 4.2.4 "vui_dma_malloc()" on page 47). For VMEbus addresses, it is safe to use page-aligned start addresses and sizes.**

RETURN VALUES

On successful completion, the number of bytes transferred is returned. Otherwise, `-1` is returned and `errno` is set to indicate the error (see the `read(2)` and `write(2)` man pages).

ERRORS

See also the `read(2)` and `write(2)` man pages

EINVAL
Invalid request or argument.

EIO
An I/O error occurred during the transaction. The fault address might also be displayed on the system console.

EXAMPLE

```
{ ...
  int vmedev;
  int rc, i;
  char *buf;

  if ((vmedev=open("/dev/vmedma32d32", O_RDWR)) == -1)
  {
      perror("open"); … /* ERROR */
  }

  /* allocate a 64K buffer and fill it */
  buf = (char*)vui_dma_malloc( vmedev, 0x10000 );
  if (buf == NULL)
```

```
  {
        perror("vui_dma_malloc"); … /* ERROR */
  }
  for ( i = 0;  i < 0x10000; i++ )
  {
        buf[i]=i;
  }

  /* seek to VMEbus address and write the buffer */
  if (lseek(vmedev, 0x60000000, SEEK_SET) == -1)
  {
        perror("lseek"); … /* ERROR */
  }
  rc = write ( vmedev, buf, 0x10000);
  if ( rc < 0)
  {
        perror("write"); … /* ERROR */
  }

  /* seek to VMEbus address and read into buffer */
  if (lseek(vmedev, 0x60000000, SEEK_SET) == -1)
  {
        perror("lseek"); … /* ERROR */
  }
  rc = read ( vmedev, buf, 0x10000 );
  if (rc <= 0)
  {
        perror("read"); … /* ERROR */
  }
  free(buf);
  (void)close(vmedev);
}
```

4.2.3 ioctl()

SYNTAX

```
#include <unistd.h>
#include <sys/vme_types.h>
#include <sys/vme.h>
#include <sys/vui.h>
int ioctl(int fildes, int request, /* arg */ ...);
```

DESCRIPTION

`ioctl()`

performs various device-specific control functions on devices. `request` and an optional third argument with varying type are passed to the file designated by `fildes` and are interpreted by the device driver. For further information see also the man pages of `ioctl(2)`.

Note: **It is strongly recommended to use VUI functions instead of `ioctl()`.**

2048369 – 0 January 2000

VARIABLES `fildes`
file descriptor of an opened VME DMA device

`request`
selects the control function to be performed and depends on the device being addressed. The following requests are defined:

```
VME_DMA_GET_STATUS
VME_DMA_INFO
```

For a description of how these requests work see the respective VUI function (`vui_dma_malloc()`,...). For examples how to use `ioctl()`, see the source code of the VUI functions.

`arg`
Parameter that might be needed by the specified device to perform the requested function. The data type of `arg` depends on the particular control request, but it is either an `int` or a pointer to a device-specific data structure.

RETURN VALUES
On successful completion, the value returned depends on the device control function, but always is a non-negative integer. Otherwise, `-1` is returned and `errno` is set to indicate the error.

ERRORS
See man page of `ioctl(2)` and VUI functions

4.2.4 vui_dma_malloc()

SYNTAX

```
#include <sys/vme_types.h>
#include <sys/vme.h>
#include <sys/vui.h>
void *vui_dma_malloc (int dev, size_t size)
```

DESCRIPTION `vui_dma_malloc()`
allocates a buffer to be used for `vmedma` DMA operations. It checks the required alignment of the hardware and calls `memalign()` instead of `malloc()`. See also the `memalign(3C)` man pages. Use `free(3C)`to release the buffer.

VARIABLES `dev`
file descriptor of an opened VME DMA device

`size`
size of the buffer to allocate

RETURN VALUES
On successful completion, the buffer address is returned. If there is no memory available or if the given file descriptor does not specify a VME

dma device, `vui_dma_malloc()` returns `NULL`. For `errno` values, see the `memalign(3C)` and `ioctl(2)` man pages.

ERRORS

See also `ioctl(2)`, memalign`(3C)`, and free`(3C)` man pages

ENOTSUP
Action not supported.

EXAMPLE

```
{ ...
  int vmedev;
  int rc, i;
  char *buf;

  if ((vmedev=open("/dev/vmedma32d32", O_RDWR)) == -1)
  {
      perror("open"); … /* ERROR */
  }

  /* allocate a 64K buffer */
  buf = (char*)vui_dma_malloc( vmedev, 0x10000 );
  if (buf == NULL)
  {
         perror("vui_dma_malloc"); … /* ERROR */
  }
  for ( i = 0;  i < 0x10000; i++ )
  {
         buf[i]=i;
  }

  /* seek toVMEbus address */
  if (lseek(vmedev, 0x60000000, SEEK_SET) == -1)
  {
         perror("lseek"); … /* ERROR */
  }

  /* write the buffer */
  rc = write ( vmedev, buf, 0x10000);
  if ( rc < 0)
  {
         perror("write"); … /* ERROR */
  }
  if (lseek(vmedev, 0x60000000, SEEK_SET) == -1)
  {
         perror("lseek"); … /* ERROR */
  }

  /* .. and read it */
  rc = read ( vmedev, buf, 0x10000 );
  if (rc <= 0)
  {
         perror("read"); … /* ERROR */
  }
  free(buf);
  (void)close(vmedev);
}
```

2049369 – 0 January 2000

4.3 vmefdma

`vmefdma` (fast DMA) avoids some software overhead typically emerging between 2 DMA transfers.

The `vmefdma` driver does not lock memory. The user process has to allocate kernel memory via VUI functions and use this memory for `read()` and `write()` calls. When re-using this memory for every request,

- the IOMMU has to be set up only once, instead of setting it up for every transfer,
- and the DMA buffer does not have to be locked because it is in kernel memory.

This saves a significant amount of time.

Note: **Depending on the system architecture, Solaris might not give the allocated memory back for normal use. Refer to the *Release Notes* for further information on allocating shared and DMA memory for the CPU board under consideration.**

Devices

By default, the driver provides access to a number of device nodes in /dev which are named as follows:
`/dev/vmefdma`<*space*><*data*>

Where <*space*> may be

- `16` for accessing data in A16 address spaces,
- `24` for accessing data in A24 address spaces,
- `32` for accessing data in A32 address spaces,
- `crcsr` for accessing data in the CR/CSR address space, and

<*data*> denotes the way the data is transferred and may be

- `d8` for 1-byte single cycles,
- `d16` for 2-byte single cycles (including 1-byte cycles),
- `d32` for 4-byte (lword) single cycles (including 1- and 2-byte transfers),
- `blt` for BLT burst cycles (including all single cycles),
- `mblt` for MBLT (D64) burst cycles (including BLT burst and single cycles), and
- `te` for 2-edge burst cycles (including all other burst and single cycles).

Routines

To access this driver the following routines are supported:

- UNIX System calls: `open()`, `close()`, `read()`, `write()`, `mmap()`, `munmap()`, `ioctl()`.
- VUI calls: `vui_fdma_malloc(), ~_free()`.

Configuration file

`/kernel/drv/vmedma.conf` is the `vmedma` configuration file. It contains a `reg` property which defines the `vmedmaxxx` device nodes created in /`dev`. For a on the format of the `reg` properties refer to section 5.2.2 "VMEbus Mappings" on page 105.
The `reg` property may be modified for accessing VMEbus address spaces not present in the default configuration.

4.3.1 open(), close()

SYNTAX

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(
          const char *path,/* path to device node */
          int oflag);/* Open Flags*/

#include <unistd.h>
int close(
          int filds);/* File handle of opened device */
```

DESCRIPTION

`open()`
obtains access to the VMEbus device and prepares it for use.

`close()`
closes a VMEbus file descriptor associated with a VMEbus device.

Both operations are similar to standard `open(2)` and `close(2)`. For further information see the respective man pages.

RETURN VALUES

On successful completion, the file descriptor is returned. Otherwise, `-1` is returned and `errno` is set to indicate the error (see the `open(2)` and `close(2)` man pages).

ERRORS

See also man pages of `open(2)` and `close(2)`

ENXIO
The minor node of the device is not supported. E.g. `vmefdma32te` is not supported for FGA5000-based CPU boards.

2049369 – 0 January 2000

4.3.2 read(), write()

SYNTAX

```
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>
#include <sys/vme_types.h>
#include <sys/vme.h>
#include <sys/vui.h>
size_t read(
          int filds, /* File handle of opened dev */
          void *buf,/* I/O address */
          size_t nbyte);/* no. of bytes to transfer */

#include <unistd.h>
size_t write(
          int filds,/* File handle of opened dev */
          const void *buf,/* I/O address */
          size_t nbyte);/* no. of bytes to transfer */
```

DESCRIPTION

`read()`
copies a block of data from the VMEbus address space to a user process buffer.

`write()`
copies a block of data from a user process buffer to the VMEbus address space.

The `read()` and `write()` function calls implement reading of or writing to a previously opened VME device. Via read and write system calls the entire 32 bit VME space is accessible. VME D32 and D16 accesses are supported.
The value of the file pointer can be set using `lseek()` and `llseek()`. With the help of these 2 function calls, the starting VME address for read or write access can be specified.

Note: **Before it is possible to read or write DMA memory via the `vmefdma` driver, it is necessary to allocate an I/O buffer via `vui_fdma_malloc()` (see page 55). Use the resulting `ioaddr` returned by `vui_fdma_malloc()` as `buf` argument for `read()` or `write()` accesses.**

Both operations are similar to standard `read(2)` and `write(2)`. For further information, see the respective man pages.

RETURN VALUES

On successful completion, the number of bytes transferred is returned. Otherwise, `-1` is returned and `errno` is set to indicate the error (see the `read(2)` and `write(2)` man pages).

ERRORS

See also the `read(2)` and `write(2)` man pages

ENXIO
The `buf` parameter does not reflect an I/O address as returned by `vui_fdma_malloc`().

EINVAL
Invalid request or argument.

EIO
An I/O error occurred during the transaction.

EXAMPLE

```
{
  ...
  int vmedev;
  int rc, i;
  u_int *uvaddr = NULL;
  u_int *cvaddr = NULL;
  ioctl_map_t fdma;

  if ((vmedev=open("/dev/vmefdma32d32", O_RDWR)) == -1)
  {
      perror("open"); … /* ERROR */
  }
  /* initialize the fdma struct */
  fdma.data_size = 100 * sizeof(int);
  fdma.prot = PROT_READ|PROT_WRITE;

  /* allocate some I/O memory */
  if ( NULL == (uvaddr = vui_fdma_malloc( vmedev, &fdma )))
  {
      perror("vui_fdma_malloc"); … /* ERROR */
  }
  /* fill the buffer */
  cvaddr = uvaddr:
  for (i = 0: i < 100; i++)
  {
          *cvaddr= i:
          cvaddr++;
  }

  /* position the file pointer */
  lseek(vmedev, 0x60000000, SEEK_SET);
  /* do the write() access */
  rc = write ( vmedev, (void *)fdma.ioaddr, * sizeof(int));
  if ( rc < 0)
  {
          perror("write"); … /* ERROR */
  }
```

2049369 – 0 January 2000

```
  /* Give the space free */
  vui_fdma_free(vmedev, &fdma, uvaddr);
  uvaddr = NULL;
  close (vmedev);
  vmedev = 0;
  if ((vmedev=open("/dev/vmefdma32d32", O_RDWR)) == -1)
  {
      perror("open"); … /* ERROR */
  }
  /* initialize the fdma struct */
  fdma.data_size = 100 * sizeof(int);
  fdma.prot = PROT_READ|PROT_WRITE;

  /* allocate some I/O memory */
  if ( NULL == (uvaddr = vui_fdma_malloc( vmedev, &fdma )))
  {
      perror("vui_fdma_malloc"); … /* ERROR */
  }
  /* position the file pointer */
  lseek(vmedev, 0x60000000, SEEK_SET);
  /* do the read() access */
  rc = read ( vmedev, (void *)fdma.ioaddr, 100 * sizeof(int)
);
  if (rc <= 0)
  {
          perror("read"); … /* ERROR */
  }
  cvaddr = uvaddr:
  for (i = 0; i < 100; i++)
  {
          if ( *cvaddr != i)
          {
           printf("WARNING:Read value failed\n");
          }
          cvaddr++;
  }
  vui_fdma_free(vmedev, &fdma, uvaddr);
  uvaddr = NULL;
  close (vmedev);
  vmedev = 0;
}
```

4.3.3 mmap(), munmap()

SYNTAX

```
#include <sys/types.h>
#include <sys/mman.h>
caddr_t mmap(
          caddr_t addr,/* has to be 0 */
          size_t len, /* block size to map in B */
          int prot, /* protection mode */
          int flags,/* flags for page handling */
          int fildes, /* file handle opened VME dev */
          off_t off);/* I/O address of DMA buffer */

#include <sys/types.h>
```

```
#include <sys/mman.h>
int munmap(
          caddr_t addr,/* usr addr for mapped VME block */
          size_t len);/* block size mapped in bytes */
```

DESCRIPTION Both operations are similar to standard `mmap(2)` and `munmap(2)`. For further information, see the respective man pages.
The `off` identified the DMA buffer to be mapped and must be set to the `ioaddr` element as returned in the `ioctl_map` structure of `vui_fdma_malloc()`.

Note: **It is strongly recommended to use `vui_fdma_malloc()` and `~_free()` instead of `ioctl()` and `mmap()` or `munmap()` (see page 55).**

RETURN VALUES

`mmap()`
On successful completion, `mmap()` returns the start address within the application's address space to which the VME device has been mapped. Otherwise, it returns `MAP_FAILED` and sets `errno` to indicate the error (see the man page for `mmap(2)`).

`munmap()`
On successful completion, `munmap()` returns `0`. Otherwise, it returns `-1` and sets `errno` to indicate the error (see the man page for `munmap(2)`).

ERRORS See also the `mmap(2)` and `munmap(2)` man pages

EFAULT
Offset, size, or alignment are erroneous.

EINVAL
Invalid request or argument, or no more ranges are available.

ENOTSUP
Action not supported.

4.3.4 ioctl()

SYNTAX

```
#include <unistd.h>
#include <sys/vme_types.h>
#include <sys/vme.h>
#include <sys/vui.h>
int ioctl(int fildes, int request, /* arg */ ...);
```

DESCRIPTION `ioctl()`
performs various device-specific control functions on devices. `request` and an optional third argument with varying type are passed to

2049369 – 0 January 2000

the file designated by `fildes` and are interpreted by the device driver. For further information see the man pages of `ioctl(2)`.

Note: **It is strongly recommended to use VUI functions instead of `ioctl()`.**

VARIABLES

`dev`
file descriptor of an opened VME device

`request`
selects the control function to be performed and depends on the device being addressed. The following requests are defined:

```
VME_FDMA_MAP
VME_FDMA_UNMAP
```

For a description of how these requests work see the respective VUI function (`vui_fdma_map()`,...). For examples how to use `ioctl()`, see the source code of the VUI functions.

`arg`
parameter that might be needed by the specified device to perform the requested function. The data type of `arg` depends on the particular control request, but it is either an `int` or a pointer to a device-specific data structure.

RETURN VALUES

On successful completion, the value returned depends on the device control function, but always is a non-negative integer. Otherwise, `-1` is returned and `errno` is set to indicate the error.

ERRORS

See man pages of `ioctl(2)` and VUI functions

4.3.5 vui_fdma_malloc(), vui_fdma_free()

SYNTAX

```
#include <sys/vme_types.h>
#include <sys/vme.h>
#include <sys/vui.h>
int vui_fdma_malloc (
          ind dev,
          ioctl_map_t *fdma)
int vui_fdma_free (
          int dev,
          ioctl_map_t *fdma,
          caddr_t vaddr)
```

DESCRIPTION

`vui_fdma_malloc()`
allocates and mmaps a buffer to be used for `vmefdma` operations.

`vui_fdma_free()`
destroys a DMA buffer (which was allocated and mapped with `vui_fdma_malloc()`) and unmaps it from the address space of the process.

VARIABLE TYPES

The structure `ioctl_map_t` is defined in `vme.h`:

```
struct ioctl_map
{
  uint_t  data_size;/* length of area to be mapped */
  vmeaddr_tvme_addr;/* not needed here */
  caddr_t kvaddr;/* reserved */
  ulong_t ioaddr;/* I/O address of buffer */
  bt_t    bt;/* zero */
  int     prot;/* Protection Mode */
  uint_t  flags;/* zero */
};
typedef struct ioctl_map ioctl_map_t;
```

`data_size`
length of area to be mapped

`kvaddr`
This element is reserved for internal use and should neither be modified nor interpreted in any kind.

`ioaddr`
returned I/O address, needed for `read()` and `write()` accesses

`prot`
protection mode (same as used for `mmap(2)`):

`PROT_READ`	Page can be read.
`PROT_WRITE`	Page can be written.
`PROT_NONE`	Page cannot be accessed.

VARIABLES

Variables for `vui_fdma_malloc()`:

`dev`
file descriptor of an opened VME device

`*fdma`
pointer to mapping structure. All mapping information is returned in this structure.

Variables for `vui_fdma_free()`:

`dev`
file descriptor of an opened VME device

`*fdma`
pointer to mapping structure. All mapping information needed to free the memory is saved within this structure.

2049369 – 0 January 2000

`vaddr`
Virtual address of the DMA buffer which has been returned by `vui_fdma_malloc()`

RETURN VALUES

`vui_fdma_malloc()`
On successful completion, `vui_fdma_malloc()` returns a virtual address where the DMA buffer can be accessed. Otherwise, `0` is returned and `errno` is set to indicate the error.

`vui_fdma_free()`
On successful completion, `VUI_OK` is returned. Otherwise, `VUI_FAIL` is returned and `errno` is set to indicate the error.

ERRORS

See also man pages of `ioctl(2)` and `mmap(2)`

EINVAL
Request or argument is invalid.

ENOMEM
Not enough memory available for DMA buffer reservation.

EXAMPLE

```
{ ...
  int vmedev;
  int rc, i;
  u_int *uvaddr = NULL;
  u_int *cvaddr = NULL;
  ioctl_map_t fdma;

  if ((vmedev=open("/dev/vmefdma32d32", O_RDWR)) == -1)
  {
          perror("open"); … /* ERROR */
  }
  /* initialize the fdma struct */
  bzero( &fdma, sizeof(fdma) );
  fdma.data_size = 0x2000;
  fdma.prot = PROT_READ|PROT_WRITE;

  /* allocate some I/O memory */
  if ( NULL == (uvaddr = vui_fdma_malloc( vmedev, &fdma )))
  {
      perror("vui_fdma_malloc"); … /* ERROR */
  }

  /* fill the buffer */
  cvaddr = uvaddr:
  for (i = 0; i < 0x2000; i++)
  {
          *cvaddr= i:
          cvaddr++;
  }
  /* position the file pointer (VMEbus address) */
  lseek(vmedev, 0x60000000, SEEK_SET);

  /* do the write() access */
```

```
  rc = write ( vmedev, (void *)fdma.ioaddr, 0x2000);
  if ( rc < 0)
  {
          perror("write"); … /* ERROR */
  }

  /* Remove the I/O buffer */
  vui_fdma_free(vmedev, &fdma, uvaddr);
  uvaddr = NULL;
  close (vmedev);
}
```

4.4 vmedvma

The vmedvma driver allows a process to set up and access on-board memory as VMEbus slave-memory. Accesses from a VMEbus master are translated into local-bus addresses, which in turn are mapped to a DVMA buffer in the on-board memory.
The on-board memory buffer has to be mapped permanently. Therefore, it can not be allocated by a user process (process memory is paged on demand), but has to be allocated within the kernel address space by the kernel. A process can access the buffer by using the `mmap()`, `read`(), and `write`() system calls.

Devices

The driver provides access to the `/dev/vmedvma`*xx* devices. The following devices are defined:

```
/dev/vmedvma24
/dev/vmedvma32
/dev/vmedvma2432
```

`vmedvma24` provides access to the shared memory in A24 space, `vmedvma32` in A32 space and `vmedvma2432` in both the A24 and the A32 space.

Routines

To access this driver the following routines are supported:

- UNIX system calls: `open()`, `close()`, `mmap()`, `munmap()`, `ioctl()`, `read()`, `write()`.
- VUI calls: `vui_slave_map()`, `~_unmap()`.

Note: **Depending on the system architecture, Solaris might not give the allocated memory back for normal use. Refer to the *Release Notes* for further information on allocating shared and DMA memory for the CPU board under consideration.**

2049369 – 0 January 2000

Default behavior

For `vmedvma` write posting is disabled per default. To change the default behavior use the bus properties which set up the slave window (see section 6 "VME Bus Properties" on page 159).
If `vme_slave_diswp_flag` is set in `/etc/system`, write posting is always disabled, regardless of other flags or VUI function calls.

Configuration file

`/kernel/drv/vmedvma.conf` is the `vmedvma` configuration file. It does not contain any configuration options.

Caution

Never change the `vmedvma` configuration file.

4.4.1 open(), close()

SYNTAX

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(
          const char *path,/* path to device node */
          int oflag);/* Open Flags*/

#include <unistd.h>
int close(int filds);/* File handle of opened device */
```

DESCRIPTION

`open()`
obtains access to a VMEbus device and prepares it for use.

`close()`
closes a VMEbus file descriptor associated with a VMEbus device.

Both operations are similar to standard `open(2)` and `close(2)`. For further information, see the respective man pages.

RETURN VALUES

On successful completion, the file descriptor is returned. Otherwise, `-1` is returned and `errno` is set to indicate the error (see the `open(2)` and `close(2)` man pages).

ERRORS

See man pages of `open(2)` and `close(2)`.

4.4.2 read(), write()

SYNTAX

```
#include <sys/types.h>
#include <unistd.h>
size_t read(
          int filds, /* File handle of opened dev */
          void *buf,/* I/O address */
          size_t nbyte);/* no. of bytes to transfer */

#include <unistd.h>
size_t write(
          int filds,/* File handle of opened dev */
          const void *buf,/* I/O address */
          size_t nbyte);/* no. of bytes to transfer */
```

DESCRIPTION

`read()`
copies a block of data from the local VME shared memory buffer to a process buffer.

`write()`
copies a block of data from a process buffer to the local VME shared memory buffer.

The `read()` and `write()` function calls implement reading of or writing to a shared memory buffer previously set up using the `vmedvma` driver.
The value of the file pointer can be set using `lseek()` and `llseek()`. It must lie within the VMEbus address range to which a shared memory buffer has been mapped (within the VME address space identified by the referenced device node). An error is returned if there is no VME shared memory mapped to the requested address.
These functions may be used for accessing a local shared memory buffer by processes other than the one which set up the mapping.

RETURN VALUES

On successful completion, the number of bytes transferred is returned. This may be less than the requested amount if the end of the accessed shared memory buffer is exceeded by the request.
Otherwise, `-1` is returned and `errno` is set to indicate the error (see the `read(2)` and `write(2)` man pages).

ERRORS

See also the `read(2)` and `write(2)` man pages

EINVAL
Invalid request or argument.

ENXIO
The current setting of the file pointer does not reference a VMEbus address to which a shared memory buffer is mapped to.

2048369 – 0 January 2000

EXAMPLE `read()` and `write()` can be demonstrated using the sample programs `shmem` and `vme_dump` which are located in the `examples` directory.

1. To start `shmem`, enter:

   ```
   shmem -n
   ```

2. Enter the information you are prompted for, e.g. address space, VMEbus address, size.

   When done, the program reports a VMEbus address where the shared memory has been mapped to and reports the syntax to be used for starting `vme_dump` in the next step.

3. Start `vme_dump` as reported by `shmem` in the previous step. In this second process `vme_dump` accesses the shared memory via `read()` and `write()` calls to the buffer.

4.4.3 mmap(), munmap()

SYNTAX

```
#include <sys/types.h>
#include <sys/mman.h>
caddr_t mmap(
          caddr_t addr,/* has to be 0 */
          size_t len, /* size of buffer */
          int prot, /* protection mode */
          int flags,/* flags for page handling */
          int fildes, /* file handle opened VME dev */
          off_t off);/* kernel virtual address of buffer */

#include <sys/types.h>
#include <sys/mman.h>
int munmap(
          caddr_t addr,/* usr addr for mapped VME block */
          size_t len);/* block size mapped in bytes */
```

DESCRIPTION `mmap()`
allows a previously allocated shared memory buffer to be mapped into an application's address space.

`munmap()`
destroys the mapping for the shared memory buffer.

The `off` parameter identifies a shared memory buffer. It must lie within the VMEbus address range to which a shared memory buffer has been mapped (within the VME address space identified by the referenced device node). An error is returned if there is no VME shared memory mapped to the requested address.
This function may be used for accessing a local shared memory buffer by processes other than the one which set up the mapping.

Note: **Memory can only be mapped as shared, so the `flags` argument has to be set to `MAP_SHARED`.**

COMPATIBILITY

The semantic of the `off` parameter has been changed with driver release 2.4. Applications which do not take the `off` parameter from the `kvaddr` element of the `ioctl_map` structure returned by `vui_smem_map()` may need to be modified.

RETURN VALUES

`mmap()`

On successful completion, `mmap()` returns the address at which the mapping was placed. Otherwise, it returns `MAP_FAILED` and sets `errno` to indicate an error (see the `mmap(2)` man page).

`munmap()`

On successful completion, the `munmap()` returns `0`. Otherwise, it returns `-1` and sets `errno` to indicate an error (see the `munmap(2)` man page).

ERRORS

See man pages of `mmap(2)` and `munmap(2)`.

ENXIO

The `off` parameter does not reference a VMEbus address to which a shared memory buffer is mapped to.

4.4.4 ioctl()

SYNTAX

```
#include <unistd.h>
#include <sys/vme_types.h>
#include <sys/vme.h>
#include <sys/vui.h>
int ioctl(int fildes, int request, /* arg */ ...);
```

DESCRIPTION

`ioctl()`

performs various device-specific control functions on devices. `request` and an optional third argument with varying type are passed to the file designated by `fildes` and are interpreted by the device driver. For further information see `ioctl(2)` man pages.

Note: **It is strongly recommended to use VUI functions instead of `ioctl()`.**

VARIABLES

`dev`

file descriptor of an opened VME device

2049369 -- 0 January 2000

`request`
selects the control function to be performed and depends on the device being addressed. The following requests are defined:

```
VME_SLAVE_MAP
VME_SLAVE_UNMAP
VME_SLAVE_SET
```

For a description of how these requests work see the respective VUI function (`vui_slave_map()`,...). For examples on how to use `ioctl()`, see the source code of the VUI functions.

`arg`
parameter that might be needed by the specified device to perform the requested function. The data type of `arg` depends on the particular control request, but it is either an `int` or a pointer to a device-specific data structure.

RETURN VALUES
On successful completion, the value returned depends on the device control function, but always is a non-negative integer. Otherwise, `-1` is returned and `errno` is set to indicate the error.

ERRORS
See the `ioctl(2)` man pages and VUI functions.

4.4.5 vui_slave_map(), vui_slave_unmap()

SYNTAX

```
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/vme_types.h>
#include <sys/vme.h>
#include <sys/vui.h>
caddr_t vui_slave_map (
          int dev, ioctl_map_t *dvma)

int vui_slave_unmap (
          int dev, ioctl_map_t *dvma, caddr_t vaddr)
```

DESCRIPTION

`vui_slave_map()`
sets up a shared memory buffer, makes it accessible from VME and optionally maps it into the process space.

`vui_slave_unmap()`
destroys the VMEbus slave window, detaches the buffer from the process address space and frees the buffer memory.

The maximum possible size of a shared memory buffer and the probability to get a VMEbus address that reflects the requested one is affected by several factors:

- the amount of physically present memory,

- the amount of contiguous kernel address space (which may decrease during system operation),
- and the amount of contiguous, properly aligned IOMMU address space (which may also decrease during system operation).

This makes predictions very hard, but as a rule of thumb one can say that the more physical memory is present and the shorter the run-time of the system, the larger the possible buffer will be.

All mappings and allocations done for a process will be freed automatically when the device is closed or the process ended.

VARIABLE TYPES

The structure `ioctl_map_t` is defined in `vme.h`:

```
struct ioctl_map
{
  uint_t  data_size;/* map In: size of buffer */
  vmeaddr_tvme_addr;/* map In/Out: VMEbus address */
  caddr_t kvaddr;/* reserved */
  ulong_t ioaddr;/* reserved */
  bt_t    bt;/* map In/Out: properties of buf. */
  int     prot;/* map In: protection mode */
  uint_t  flags;/* map In: flags how to set up buf. */
  vmeaddr_tdc_vmeaddr;/* map Out: Decoded VMEbus address */
  size_t  dc_size;/* map Out: Decoded size */
};
typedef struct ioctl_map ioctl_map_t;
```

`data_size`

In: size of the shared memory buffer

`vme_addr`

In: VMEbus address for the shared memory buffer

Out: resulting VMEbus address of the shared memory buffer. The input address is only a suggestion. The output address is the actually used address.

Note: **Depending on the `flags` argument, the actual VMEbus address may differ from the requested one. If it differs, this accomplishes for a VME interface hardware requiring alignments for which the Solaris DMA mechanism is not designed.**

`kvaddr`
`ioaddr`

These elements are reserved for internal use and should neither be modified nor interpreted in any kind.

`bt`

In: flag for the bus property. All available flags are defined in `vme_types.h` (see section 6 "VME Bus Properties" on page 159).

2049369 – 0 January 2000

They can be logically OR-ed. See the *Release Notes* for flags which are relevant for the CPU board under consideration.

Out: flags which actually are used.

`prot`
In: The protection mode is the same as used by `mmap(2)`:

`PROT_READ`	Buffer can be read.
`PROT_WRITE`	Buffer can be written.
`PROT_NONE`	Buffer can not be accessed.

`flags`
Flags that affect the way how the shared memory is set up:

`SMEM_PADDR`	reserved for future extensions, currently unused.
`SMEM_VADDR`	If this flag is set, the standard method of setting up the shared memory buffer is used. Due to hardware limitations, the VMEbus address to which the shared memory is actually mapped might differ from the requested one. Refer to the *Release Notes* for information on address offsets which are to be expected for the hardware under consideration. Currently this flag must be set. It may be combined with the flags described below.
`SMEM_FIXED`	If this flag is set, the VMEbus nexus driver sets up the shared memory at exactly the requested VMEbus address, provided that the requested VMEbus address is aligned to page boundary. The decoded VMEbus address range might be larger than the shared memory address range. Using this flag might fragment system resources more than not using the flag. See the *Release Notes* whether this flag is supported for the CPU board under consideration.
`SMEM_DONTMAP`	Set up the shared memory and map it to VME, but don't map it to the process address space. In this case, the only way to access the buffer locally is to use the `read()` and `write()` interface.

`dc_vmeaddr`, `dc_size`
Out: When setting up the VME slave window needed to access the onboard memory, it may be necessary to set up a larger window than the requested one. The address range of the actually used window is re-

ported by dc_vmeadd and dc_size. The VMEbus interface chip responds to all master accesses within this range, so make sure that it does not conflict with other bus participants.

COMPATIBILITY

The semantic of the kvaddr and ioaddr elements has been changed with driver version 2.4. Applications that use these values for any other purposes or modify these values may need to be changed.

VARIABLES

Variables for vui_slave_map():

dev
file descriptor of an opened VME device

*dvma
pointer to mapping structure

Variables for vui_slave_unmap():

dev
file descriptor of an opened VME device

*dvma
pointer to mapping structure

vaddr
Virtual address of the slave memory buffer which has been returned by vui_slave_map(). This parameter is ignored if the request flag SMEM_DONTMAP is set.

RETURN VALUES

vui_slave_map()
This function returns 0 if an error occurred and errno is set. On success, it returns a virtual address where the slave memory can be accessed.

If the request flag SMEM_DONTMAP has been set, vui_slave_map() returns SMEM_MAPPED to indicate success.

vui_slave_unmap()
On successful completion, VUI_OK is returned. Otherwise, VUI_FAIL is returned and errno is set to indicate the error.

ERRORS

See also ioctl(2) man pages

EAGAIN
At the time of execution the action cannot be done, e.g. because no resources are available.

EFAULT
Offset, size, or alignment are erroneous.

EINVAL
Invalid request or argument.

20493G9 – 0 January 2000

ENOMEM
Possible causes:

– The composite size of `dvma.data_size` plus the lengths of all previous mappings via `mmap()` exceeds `RLIMIT_VMEM` (see the man pages of `getrlimit(2)`).

– Not enough I/O memory available for the mapping.

ENOTSUP
Action not supported.

EXAMPLE

```
{
  ioctl_map_t dvma;
  int fd;
  char *bufp;
  int i;
  if (-1 == (fd = open("/dev/vmedvma24", O_RDWR )))
  {
         perror("open"); … /* ERROR */
  }
  /* allocate and map 1 MB, give VMEbus address 0 as hint.
   * enable write posting. */
  dvma.data_size  = 0x100000;
  dvma.vme_addr   = 0x0;
  dvma.flags      = SMEM_VADDR;
  dvma.bt         = VME_BT_WP;
  bufp = (char*)vui_slave_map( fd, &dvma );

  if (bufp == NULL)
  {
         perror( "vui_slave_map" ); … /* ERROR */
  }

  /* report values */
  printf( "--> Slave window at VME 0x%x\n",
                 (u_int)dvma.vme_addr );
  printf( "--> Decoded slave range: 0x%x + 0x%x\n",
                 (u_int)dvma.dc_vmeaddr,
                 (u_int)dvma.dc_size );

  ...
  wait for someone to write data to the buffer
  ...

  /* print some bytes */
  for (i = 0; i < 16; i++)
         printf( "%02x ", (int)(bufp[i])&0xff );
  printf( "\n" );
  /* destroy the slave memory and unmap it */
  printf( "-> Unmapping buffer..." );
  vui_slave_unmap( fd, &dvma, bufp );
  ...
  close(fd);
}
```

4.5 vmectl

The vmectl driver provides various control and debug options. It also supports mailboxes and hardware signals like abort, sysfail or acfail. Furthermore, it can be used to obtain information about the CPU board and the VME interface chip used.

Devices

The driver provides access to the /dev/vmectl device.

Routines

To access this driver the following routines are supported:

- UNIX System calls: open(), close(), ioctl().
- VUI calls:
  vui_abort_wait(), ~_signal(),
  vui_acfail_wait(), ~_signal(),
  vui_arb_mode_set(), ~_get(),
  vui_board(),
  vui_bus_rel_mode_set(), ~_get(),
  vui_bus_req_level_set(), ~_get(),
  vui_bus_req_mode_set(), ~_get(),
  vui_interface(),
  vme_intr_generate(),
  vui_mbox_info(), ~_set(), ~_remove(), ~_wait(),
  vui_reg_base_set(), ~_get(),
  vui_reg_read(), ~_write(),
  vui_reset(),
  vui_sysfail_assert, ~_deassert,
  vui_(n)sysfail_wait(), ~_signal().

Configuration file

/kernel/drv/vmectl.conf is the vmectl configuration file. It contains one configuration option which controls the behavior of vui_acfail_wait(), ~_signal(), vui_sysfail_wait(), ~_signal(), vui_nsysfail_wait(), and ~_signal(). For further information refer to the respective comment in vmectl.conf.

4.5.1 open(), close()

SYNTAX

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(
          const char *path,/* path to device node */
          int oflag);/* Open Flags*/

#include <unistd.h>
int close(int filds);/* File handle of opened device */
```

2049369 – 0 January 2000

DESCRIPTION

`open()`
obtains access to a VMEbus device and prepares it for use.

`close()`
closes a VMEbus file descriptor associated with a VMEbus device.

Both operations are similar to standard `open(2)` and `close(2)`. For further information, see the respective man pages.

RETURN VALUES

On successful completion, the file descriptor is returned. Otherwise, `-1` is returned and `errno` is set to indicate the error (see the `open(2)` and `close(2)` man pages).

ERRORS

See the `open(2)` and `close(2)` man pages

4.5.2 ioctl()

SYNTAX

```
#include <unistd.h>
#include <sys/vme_types.h>
#include <sys/vme.h>
#include <sys/vui.h>
int ioctl(int fildes, int request, /* arg */ ...);
```

DESCRIPTION

`ioctl()`
performs various device-specific control functions on devices. `request` and an optional third argument with varying type are passed to the file designated by `fildes` and are interpreted by the device driver. For further information see the `ioctl(2)` man pages.

Note: **It is strongly recommended to use VUI functions instead of `ioctl()`.**

VARIABLES

`dev`
file descriptor of an opened VME device

`request`
selects the control function to be performed and depends on the device being addressed. The following requests are defined:

```
VME_REG_READ
VME_REG_WRITE
VME_REG_BASE_SET
VME_REG_BASE_GET
VME_ARB_MODE_SET
VME_ARB_MODE_GET
VME_BRL_SET
VME_BRL_GET
VME_BRM_SET
```

```
VME_BRM_GET
VME_BREL_SET
VME_BREL_GET
VME_INTR_GENERATE
VME_MBOX_SET
VME_MBOX_GET
VME_MBOX_ENABLE
VME_MBOX_DISABLE
VME_MBOX_WAIT
VME_BOARD
VME_INTERFACE
VME_ABORT_INTR
VME_ACFAIL_INTR
VME_SYSFAIL_INTR
VME_RESET
```

For a description of how these requests work see the respective VUI function (`vui_reg_read()`,...). For examples how to use `ioctl()`, see the source code of the VUI functions.

`arg`
parameter that might be needed by the specified device to perform the requested function. The data type of `arg` depends on the particular control request, but it is either an `int` or a pointer to a device-specific data structure.

RETURN VALUES
On successful completion, the value returned depends on the device control function, but always is a non-negative integer. Otherwise, `-1` is returned and `errno` is set to indicate the error.

ERRORS
See the `ioctl(2)` man pages

4.5.3 vui_abort_signal(), vui_abort_wait()

SYNTAX

```
#include <sys/vme_types.h>
#include <sys/vme.h>
#include <sys/vui.h>
int vui_abort_signal (int dev, int signal)
int vui_abort_wait (int dev)
```

DESCRIPTION

`vui_abort_signal()`
sets the signal which shall be sent to the user application when the front-panel abort key is triggered.

`vui_abort_wait()`
waits for the abort key to be triggered.

VARIABLES
Variables for `vui_abort_signal()`:
`dev`
file descriptor of an opened VME device

`signal`
signal to be sent to the user application when the front-panel abort key is triggered. If set back to `0`, sending the signal is stopped.

Variables for `vui_abort_wait()`:
`dev`
file descriptor of an opened VME device

Note: **Not all signals can be sent to an application. For a list of possible signals, see the `proc_signal(9F)` man pages.**

RETURN VALUES
On successful completion, `VUI_OK` is returned by both functions. Otherwise, `VUI_FAIL` is returned and `errno` is set to indicate the error.

ERRORS
See also `ioctl(2)` man pages

EAGAIN
At the time of execution the action cannot be done, e.g. because no resources are available.

ECANCELED
A timeout occurred.

EINTR
The process has been interrupted by a signal while waiting.

EINVAL
Invalid request or argument, or no more ranges are available.

ENOTSUP
Action not supported.

EXAMPLE 1

```
{
  int vmedev;
  ...
  if ((vmedev=open("/dev/vmectl", O_RDWR)) == -1)
  {
          perror("open"); … /* ERROR */
  }
  /* wait for the ABORT switch to be triggered */
  vui_abort_wait (vmedev);
  ...
  (void)close(vmedev);
  ...
}
```

EXAMPLE 2

```
{
  int vmedev;
  ...
  if ((vmedev=open("/dev/vmectl", O_RDWR)) == -1)
  {
          perror("open"); … /* ERROR */
  }
  ...
  /* prepare a signal to be sent when the ABORT switch is
triggered */
  sigset (SIGINT, aborthdl );
  vui_abort_signal(vmedev, SIGINT);
  ...
  vui_abort_signal(vmedev, 0);
  (void)close(vmedev);
  ...
}
```

4.5.4 vui_acfail_signal(), vui_acfail_wait()

SYNTAX

```
#include <sys/vme_types.h>
#include <sys/vme.h>
#include <sys/vui.h>
int vui_acfail_wait (int dev)
int vui_acfail_signal (int dev, int signal)
```

DESCRIPTION

`vui_acfail_wait()`
waits for the VME ACFAIL* to be asserted.

`vui_acfail_signal()`
sets the signal which shall be sent to the user application when VME ACFAIL* is asserted.

VARIABLES

Variables for `vui_acfail_wait()`:

`dev`
file descriptor of an opened VME device

Variables for `vui_acfail_signal()`:

`dev`
file descriptor of an opened VME device

`signal`
signal to be sent to the user application when VME ACFAIL* is asserted. By default the `vmectl` driver is configured to detect only transitions of the ACFAIL line from high to low, but not the current state of the ACFAIL line. This behavior can be changed by modifying the driver configuration file. For further information refer to the respective comment in `vmectl.conf`.

If set back to `0`, sending the signal is stopped.

2049369 – 0 January 2000

Note: **Not all signals can be sent to an application. For a list of possible signals, see the `proc_signal(9F)` man page.**

RETURN VALUES

On successful completion, `VUI_OK` is returned. Otherwise, `VUI_FAIL` is returned and `errno` is set to indicate the error.

ERRORS

See also `ioctl(2)` man pages

EAGAIN
At the time of execution the action cannot be done, e.g. because another process is using the ACFAIL functions.

ECANCELED
A timeout occurred.

EINTR
The process has been interrupted by a signal while waiting.

EINVAL
Invalid request or argument, or no more ranges are available.

ENOTSUP
Action not supported.

EXAMPLE 1

```
{
  int vmedev;
  if ((vmedev=open("/dev/vmectl", O_RDWR)) == -1)
  {
          perror("open"); … /* ERROR */
  }
  /* wait for the ACFAIL to be triggered */
  vui_acfail_wait (vmedev);
  ...
  (void)close(vmedev);
  ...
}
```

EXAMPLE 2

```
{
  int vmedev;
  if ((vmedev=open("/dev/vmectl", O_RDWR)) == -1)
  {
          perror("open"); … /* ERROR */
  }
  ...
  /* prepare a signal to be sent when ACFAIL is triggered */
  sigset( SIGINT, acfail_hdl );
  vui_acfail_signal(vmedev, SIGINT);
  ...
  vui_acfail_signal(vmedev, 0);
  ...
  (void)close(vmedev);
  ...
```

```
}
```

4.5.5 vui_arb_mode_set(), vui_arb_mode_get()

SYNTAX

```
#include <sys/vme_types.h>
#include <sys/vme.h>
#include <sys/vui.h>
int vui_arb_mode_set (int dev, int arb)
int vui_arb_mode_get (int dev, int *arb)
```

DESCRIPTION

`vui_arb_mode_set()`
sets the arbitration mode of the local VMEbus arbiter.

`vui_arb_mode_get()`
returns the arbitration mode the local VMEbus arbiter is currently running in.

VARIABLE

Variables for `vui_arb_mode_set()`:

`dev`
file descriptor of an opened VME device

`arb`
can have the following values:

`VME_ARB_SGL`	single level arbiter on level 3
`VME_ARB_RR`	round robin arbiter
`VME_ARB_PRI`	priority arbiter with level 3 being the highest priority level
`VME_ARB_PRIRR`	priority round robin arbiter
`VME_ARB_OFF`	the board is not system controller (slot-1 device)

The values are defined in `vme_types.h`. The arbitration mode can not be set or requested if the local CPU board is not the VMEbus system controller (VMEbus slot 0).

Variables for `vui_arb_mode_get()`:

`dev`
file descriptor of an opened VME device

`*arb`
pointer to the current arbitration mode. The current arbitration mode is returned within this pointer. For possible arbitration modes see above.

RETURN VALUES

On successful completion, `VUI_OK` is returned. Otherwise, e.g. if CPU is not arbiter, `VUI_FAIL` is returned and `errno` is set to indicate the error.

ERRORS See also `ioctl(2)` man pages

EINVAL
Invalid argument.

ENOTSUP
Action not supported.

EXAMPLE
```
{ ...
  int vmedev;
  int arb;

  if ((vmedev=open("/dev/vmectl", O_RDWR)) == -1)
  {
          perror("open"); … /* ERROR */
  }
  vui_arb_mode_set( vmedev, VME_ARB_PRIRR );
  vui_arb_mode_get( vmedev, &arb );
  ...
  (void)close(vmedev);
  ...
}
```

4.5.6 vui_board()

SYNTAX
```
#include <sys/vme_types.h>
#include <sys/vme.h>
#include <sys/vui.h>
int vui_board(
          int dev,
          char *name,/* board name max. 32 char */
          short *rel);/* release */
```

DESCRIPTION `vui_board()`
returns the CPU board's name and the LCA revision.

VARIABLES `dev`
file descriptor of an opened VME device

`*name`
pointer to a buffer, the CPU board's name is copied to

`*rel`
pointer to the revision number

RETURN VALUES On successful completion, `VUI_OK` is returned. Otherwise, `VUI_FAIL` is returned and `errno` is set to indicate the error.

ERRORS See `ioctl(2)` man pages

EXAMPLE

```
{
  int fd;
  int id;
  char name[32];
  ...
  if (-1 == (fd = open("/dev/vmectl", O_RDWR )))
  {
          perror( "open" ); … /* ERROR */
 … /* ERROR */}

  if (-1 == vui_board( fd, name, &id ))
  {
          perror( "vui_board" ); … /* ERROR */
  }

  printf( "-> Board name: %s\n", name );
  printf( "-> Board rev.: %d\n", (int)id );
}
```

4.5.7 vui_bus_rel_mode_set(), vui_bus_rel_mode_get()

SYNTAX

```
#include <sys/vme_types.h>
#include <sys/vme.h>
#include <sys/vui.h>
int vui_bus_rel_mode_set (int dev, int brel)
int vui_bus_rel_mode_get (int dev, int *brel)
```

DESCRIPTION

`vui_bus_rel_mode_set()`
sets the VMEbus release mode for future master accesses.

`vui_bus_rel_mode_get()`
returns the current release mode.

VARIABLES

Variables for `vui_bus_rel_mode_set()`:

`dev`
file descriptor of an opened VME device

`brm`
specifies the VMEbus release mode, defined in `vme_types.h`:

`VME_BRL_ROR`	Release on request
`VME_BRL_ROC`	Release on clear
`VME_BRL_RAT`	Release after timeout
`VME_BRL_RWD`	Release when done

2049369 – 0 January 2000

Variables for `vui_bus_rel_mode_get()`:

`dev`
file descriptor of an opened VME device

`*brm`
pointer to the current VMEbus release mode. This pointer returns the current release mode. For possible values of the release mode see above.

RETURN VALUES
On successful completion, `VUI_OK` is returned. Otherwise, `VUI_FAIL` is returned and `errno` is set to indicate the error.

ERRORS
See also `ioctl(2)` man pages

EINVAL
Invalid argument.

ENOTSUP
Action not supported.

EXAMPLE

```
{
  ...
  int vmedev;
  int brel;

  if ((vmedev=open("/dev/vmectl", O_RDWR)) == -1)
  {
         perror("open"); … /* ERROR */
  }
  vui_bus_rel_mode_set( vmedev, VME_BRL_ROR );
  vui_bus_rel_mode_get( vmedev, &brel );
  printf("->bus release mode: %d\n", brel);
  ...
  (void)close(vmedev);
  ...
}
```

4.5.8 vui_bus_req_level_set(), vui_bus_req_level_get()

SYNTAX

```
#include <sys/vme_types.h>
#include <sys/vme.h>
#include <sys/vui.h>
int vui_bus_req_level_set (int dev, int brl)
int vui_bus_req_level_get (int dev, int *brl)
```

DESCRIPTION

`vui_bus_req_level_set()`
sets the bus request level at which all future master accesses will be performed.

`vui_bus_req_level_get()`
returns the current bus request level.

VARIABLE

Variables for `vui_bus_req_level_set()`:

`dev`
file descriptor of an opened VME device

`brl`
bus request level: `0` to `3`

Variables for `vui_bus_req_level_get()`:

`dev`
file descriptor of an opened VME device

`*brl`
pointer to bus request level. This pointer returns the current bus request level. Possible values are `0` to `3`.

RETURN VALUES

On successful completion, `VUI_OK` is returned. Otherwise, `VUI_FAIL` is returned and `errno` is set to indicate the error.

ERRORS

See also `ioctl(2)` man pages

EINVAL
Invalid request or argument, or no more ranges are available.

ENOTSUP
Action not supported.

EXAMPLE

```
{
  ...
  int vmedev;
  int brl;

  if ((vmedev=open("/dev/vmectl", O_RDWR)) == -1)
  {
          perror("open"); … /* ERROR */
  }
  vui_bus_req_level_set( vmedev, 2 );
  vui_bus_req_level_get( vmedev, &brl );
  printf("->bus request level: %d\n", brl);
  ...
  (void)close(vmedev);
  ...
}
```

4.5.9 vui_bus_req_mode_set(), vui_bus_req_mode_get()

SYNTAX

```
#include <sys/vme_types.h>
#include <sys/vme.h>
#include <sys/vui.h>
int vui_bus_req_mode_set (int dev, int brm)
int vui_bus_req_mode_get (int dev, int *brm)
```

DESCRIPTION

`vui_bus_req_mode_set()`
sets the VMEbus request mode for future master transfers.

`vui_bus_req_mode_get()`
returns the current VMEbus request mode.

VARIABLES

Variables for `vui_bus_req_mode_set()`:

`dev`
file descriptor of an opened VME device

`brm`
specifies the VMEbus request mode, defined in `vme_types.h`:

`VME_BRQ_FAIR`	fair request mode
`VME_BRQ_DEMAND`	demand request mode

Variables for `vui_bus_req_mode_get()`:

`dev`
file descriptor of an opened VME device

`*brm`
pointer to the VMEbus request mode. This pointer returns the current bus request mode. For possible values see above.

RETURN VALUES

On successful completion, `VUI_OK` is returned. Otherwise, `VUI_FAIL` is returned and `errno` is set to indicate the error.

ERRORS

See also `ioctl(2)` man pages

EINVAL
Invalid request or argument, or no more ranges are available.

ENOTSUP
Action not supported.

EXAMPLE

```
{
  ...
  int vmedev;
  int brm;

  if ((vmedev=open("/dev/vmectl", O_RDWR)) == -1)
  {
         perror("open"); … /* ERROR */
  }
  vui_bus_req_mode_set( vmedev, VME_BRQ_FAIR );
  vui_bus_req_mode_get( vmedev, &brm );
  printf("->bus request mode: %d\n", brm);
  ...
  (void)close(vmedev);
  ...
}
```

4.5.10 vui_interface()

SYNTAX

```
#include <sys/vme_types.h>
#include <sys/vme.h>
#include <sys/vui.h>
int vui_interface(
          char *name,/* name string, max. 32 char */
          short *rev );/* LCA revision */
```

DESCRIPTION

`vui_interface()`
returns the interface name and the LCA revision.

VARIABLES

`dev`
file descriptor of an opened VME device

`name`
pointer to a buffer, the interface name is copied to

`rev`
pointer to the revision number

RETURN VALUES

On successful completion, `VUI_OK` is returned by all functions. Otherwise, `VUI_FAIL` is returned and `errno` is set to indicate the error.

ERRORS

See `ioctl(2)` man pages

EXAMPLE

```
{
  int fd;
  int id;
  char name[32];

  if ((fd=open("/dev/vmectl", O_RDWR)) == -1)
  {
         perror("open"); … /* ERROR */
  }
```

2049369 – 0 January 2000

```
  ...
  if (-1 == vui_interface( fd, name, &id ))
  {
          perror( "vui_interface" ); … /* ERROR */
  }

  printf( "-> Interface name: %s\n", name );
  printf( "-> Interface rev.: %d\n", (int)id );
  ...
  close (fd);
}
```

4.5.11 vui_intr_generate()

SYNTAX

```
#include <sys/vme_types.h>
#include <sys/vme.h>
#include <sys/vui.h>
int vui_intr_generate (int dev, ioctl_irq_t *intr)
```

DESCRIPTION

`vui_intr_generate()`
triggers VMEbus interrupts. The function waits at most 1 second (uninterruptable) for the interrupt to be acknowledged, otherwise an error is returned.

Note: **The generation of interrupts is hardware dependent. Therefore, refer to the *Release Notes* whether this feature is supported on the CPU board under consideration.**

VARIABLE TYPES

The structure `ioctl_irq_t` is defined in `vme.h`:

```
struct ioctl_irq
{
  int     prop;   /* not needed here*/
  int     sig;    /* not needed here*/
  int     level;  /* VMEbus interrupt level */
  int     vector; /* VMEbus interrupt vector */
};
typedef struct ioctl_irq      ioctl_irq_t;
```

`level`
interrupt level to be triggered: `1` … `7`

`vect`
interrupt vector to be triggered: `0`…`255`

VARIABLES

`dev`
file descriptor of an opened VME device

`intr`
points to interrupt definition structure.

RETURN VALUES

On successful completion, `VUI_OK` is returned. Otherwise, `VUI_FAIL` is returned and `errno` is set to indicate the error.

ERRORS

See also `ioctl(2)` man pages

EACCES
The `/dev/vmectl` device was not opened for write access.

EAGAIN
The interrupt could not be generated because another interrupt triggered by the local CPU on this level is not acknowledged yet.

ECANCELED
The IACK cycle did not finish within one second.

EINVAL
Invalid request or argument.

EXAMPLE

```
{
  int vmedev;
  ioctl_irq_t intr;
  int retry;
  int error = 1;

  if ((vmedev=open("/dev/vmectl", O_RDWR)) == -1)
  {
          perror("open");
          exit(1);
  }
  intr.level = 5;
  intr.vect = 0x5c;

  /* Attempt to send interrupt. Retry ten times.
   */
  for (retry = 0; retry < 10; retry++)
  {
          if (vui_intr_generate(vmedev, &intr) == VUI_OK);
          {
                  /* Success. Exit retry loop
                   */
                  error = 0;
                  break;
          }
          if (errno == ECANCELED)
          {
                  /* Timeout after interrupt was sent.Maybe
                   * the interrupt handler is very busy. It
                   * might also be a hardware failure.
                   * Exit retry loop.
                   */
                  error = 0;
                  fprintf(stderr, "IACK timed out\n" );
                  break;
          }
```

2049369 – 0 January 2000

```
                else if (errno != EAGAIN)
                {
                        /* fatal error
                         */
                        perror("vui_intr_generate");
                        break;
                }

                fprintf(stderr,
                        "Old IRQ still pending, retrying\n");
        }

        if (error)
        {
                fprintf(stderr, "Error sending interrupt\n" );
                exit(1);
        }
        (void)close(vmedev);
}
```

4.5.12 vui_mbox_info()

SYNTAX

```
#include <sys/vme_types.h>
#include <sys/vme.h>
#include <sys/vui.h>
int vui_mbox_info (int dev, ioctl_mbox_info_t *mbox)
```

DESCRIPTION

`vui_mbox_info()`
returns information about the mailbox capabilities and the current allocation status.

Mailboxes are resources in the VMEbus bridge which trigger a local interrupt when being accessed from the VMEbus.

VARIABLE TYPES

The structure `ioctl_mbox_info_t` is defined in `vme.h`:

```
struct ioctl_mbox_info
{
  int     nmbox;  /* no. of mailboxes */
  int     nmbox_inuse;/* mailboxes in use */
  bt_t    mbox_bt;/* avail. bus prop. */
  vmeaddr_tmbox_offset_def;/* address offset */
  vmeaddr_tmbox_offset_mask;/* address mask */
  uint_t  mbox_access;/* access method(s) */
};
typedef struct ioctl_mbox_info ioctl_mbox_info_t;
```

`nmbox`
total number of available mailboxes.

`nmbox_inuse`
number of mailboxes in use.

`mbox_bt`
VMEbus properties for the available mailboxes (see section 6 "VME Bus Properties" on page 159).

`mbox_offset_mask`
A bit mask that denotes the address bits which are compared by the hardware for detecting a mailbox access.

`mbox_offset_def`
an offset from a VMEbus address which is aligned to `mbox_offset_mask`. The resulting address is the address where the mailbox is accessible. This offset usually results from the fact that mailboxes are in fact registers in the VMEbus bridge hardware which are made accessible from VME. In this case, the mask results from the setting/alignment of the register access image and the offset from the mailbox register within the complete register set.

`mbox_access`
specifies the kind of access which triggers a mailbox interrupt:

`VME_MB_RD`	trigger by read access.
`VME_MB_WR`	trigger by write access.
`VME_MB_RDWR`	trigger by both read or write access.

VARIABLES

`dev`
file descriptor of an opened VME device

`mbox`
points to mailbox definition structure `ioctl_mbox_info_t`

RETURN VALUES
On successful completion, `VUI_OK` is returned and the ioctl_mbox_t structure is filled with the known values. Otherwise, `VUI_FAIL` is returned and `errno` is set to indicate the error.

ERRORS
See also `ioctl(2)` man pages

ENOTSUP
Hardware does not support mailboxes.

4.5.13 vui_mbox_set(), vui_mbox_remove()

SYNTAX

```
#include <sys/vme_types.h>
#include <sys/vme.h>
#include <sys/vui.h>
int vui_mbox_set (int dev, ioctl_mbox_t *mbox);
int vui_mbox_remove (int dev, int mbox_num);
```

2048369 – 0 January 2000

DESCRIPTION

`vui_mbox_set()`
allocates a mailbox for use by the calling process. A mailbox may be operated in 2 modes:

- In "buffered mode", the mailbox behaves like a semaphore counting a mailbox access counter. The access counter is increased upon each access (V operation), and decreased when a process issues a wait operation (P operation).
  A buffered mailbox can be in state "disabled", i.e. the counter is not increased by an access, or "enabled". After a mailbox has sucessfully been initialized by `vui_mbox_set()`, it is disabled. It can be enabled by calling `vui_mbox_wait`() or `vui_mbox_control()`.
- In "non-buffered" mode, the mailbox is enabled if and only if a process is waiting for it. Intermediate accesses to the mailbox have no effect.

`vui_mbox_remove()`
releases (destroys) a mailbox.

VARIABLE TYPES

The mailbox request structure `ioctl_mbox_t` is defined in `vme.h`:

```
struct ioctl_mbox
{
  int     mbox_num;/* Out: Mailbox id */
  vmeaddr_tmbox_addr;/* Out: VMEbus address */
  bt_t    mbox_vme_space;/* In/Out: VME properties */
  vmeaddr_tmbox_vme_min;/* In: low mbox addr. */
  vmeaddr_tmbox_vme_max;/* In: upper mbox addr. */
  uint_t  mbox_access;/* In/Out: mailbox modes */
};
typedef struct ioctl_mbox ioctl_mbox_t;
```

`mbox_num`
`vui_mbox_set`() sets this value to the ID of the allocated mailbox.

`mbox_addr`
Gets set with the VMEbus address of the allocated mailbox.

`vme_space`
This bit set specifies the VMEbus bus properties of a requested mailbox (see section 6 “VME Bus Properties” on page 159).

Upon successful return, `vme_space` contains the actual VME bus properties of the allocated mailboxes. All property bits set in the request are guaranteed to be satisfied. `vui_mbox_set`() might set additional properties if hardware requires it.

`vui_mbox_set`() returns an error if property bits are set which can not be satisfied by hardware.

`mbox_vme_min, mbox_vme_max`
Defines a lower and inclusive upper address boundary for a mailbox request. `vui_mbox_set()` returns an error if it is not possible to allocate a mailbox within this address range.

`mbox_access`
specifies further properties of the mailbox to be allocated:

`VME_MB_RD, or` `VME_MB_WR, or` `VME_MB_RDWR`	Mailbox is triggered upon reading, writing to its address (or both) respectively. Not setting these flags is valid (a default value will be selected and returned).
`VME_MB_BUFFER`	Indicates that the mailbox shall be requested in buffered mode. If this flag is not set, the mailbox is requested in non-buffered mode.

Upon successful return, `vui_mbox_set()` returns the actual access type(s) in `mbox_access`. If set, access types `VME_MB_RD`, `~WR` and `~RDWR` are guaranteed to remain valid, but they might get extended (e.g. if read access was requested, a mailbox might be triggered by read and write accesses). If a specified access mode cannot be granted, an error is returned.

VARIABLES

Variables for `vui_mbox_set()`:
`dev`
file descriptor of an opened vmectl device

`*mbox`
points to mailbox definition structure.

Variables for `vui_mbox_remove()`:
`dev`
file descriptor of an opened vmectl device

`mbox_num`
mailbox-ID which has been returned in the mailbox definition structure by `vui_mbox_set()`

RETURN VALUES

On successful completion, `VUI_OK` is returned. Otherwise, `VUI_FAIL` is returned and `errno` is set to indicate the error.

ERRORS

See also `ioctl(2)` man pages

EAGAIN
At execution time the action cannot be done, e.g. because no resources are available.

EINVAL
Invalid request or argument.

2049369 – 0 January 2000

ENOTSUP
Action not supported.

EXAMPLE

```
{
  int fd;
  ioctl_mbox_t mbox;
  ...
  if (-1 == (fd = open("/dev/vmectl", O_RDWR )))
  {
    perror( "open" ); … /* ERROR */
  }

  mbox.mbox_vme_space=VME_BT_A16 | VME_BT_D8;
  mbox.mbox_vme_min=0;
  mbox.mbox_vme_max=0xffff;
  mbox.mbox_access=VME_MB_BUFFER;

  if (-1 == vui_mbox_set( fd, &mbox ))
  {
    perror( "vui_mbox_set" ); … /* ERROR */
  }
  printf( "--> Mailbox id:      %d\n",  mbox.mbox_num );
  printf( "--> Mailbox address: 0x%x\n",
          (u_int)mbox.mbox_addr );
  printf( "--> Mailbox access:  0x%x\n", mbox.mbox_access );
  ...
  close(mbox);
}
```

4.5.14 vui_mbox_wait()

SYNTAX

```
#include <sys/vme_types.h>
#include <sys/vme.h>
#include <sys/vui.h>
int vui_mbox_wait (int dev, int mboxnum)
```

DESCRIPTION

`vui_mbox_wait()`
Waits for a mailbox being accessed. The exact behavior depends on whether the mailbox has been initialized in buffered or non-buffered mode (see "vui_mbox_set(), vui_mbox_remove()" on page 84).
If the mailbox is operated in buffered mode and it has not yet been enabled by vui_mbox_control(), it will automatically be enabled by the vui_mbox_wait() call.

VARIABLES

`dev`
file descriptor of an opened VME device

`mboxnum`
mailbox ID – set when returning from `vui_mbox_set()`

RETURN VALUES
On successful completion, `VUI_OK` is returned. Otherwise, `VUI_FAIL` is returned and `errno` is set to indicate the error.

ERRORS See also `ioctl(2)` man pages

EINVAL
Invalid request or argument.

`EAGAIN`
Another process is already waiting at the specified mailbox. This is currently not supported.

ENOSPC
The mailbox is used in buffered mode and an overflow of the mailbox counter has occured. If this happens, the mailbox is automatically disabled. Use the vui_mbox_control() function to reset its state.

4.5.15 vui_mbox_control()

SYNTAX
```
#include <sys/vme_types.h>
#include <sys/vme.h>
#include <sys/vui.h>
int vui_mbox_control( int dev, int mboxnum, int ctlop,
                      caddr_t arg);
```

DESCRIPTION This function performs various control operations on a mailbox which has been initialized by `vui_mbox_set()`.

VARIABLES `dev`
file descriptor of an opened vmectl device

`mboxnum`
mailbox ID – set when returning from `vui_mbox_set()`

`ctlop, arg`
Defines the control operation and an argument to the control operation:

Table 4 Mailbox Control operations

ctlop	arg	Description
VUI_MBOX_ENA	0	Enables a mailbox. This has no effect if the mailbox is already enabled.
	VUI_MBOX_RESET	Resets the mailbox counter to zero and enables the mailbox.

2049369 – 0 January 2000

Table 4 Mailbox Control operations (cont.)

ctlop	arg	Description
VUI_MBOX_DISA	0	Disables a buffered mailbox. Further accesses will not increase the mailbox counter.
VUI_MBOX_CNTGET	int *cntr	Stores the current value of the mailbox access counter in *cntr. The counter is not altered.

The control operations VUI_MBOX_ENA, VUI_MBOX_DISA and VUI_MBOX_CNTGET are only valid on buffered mailboxes.
In case of a mailbox counter overflow detected by vui_mbox_wait(), the counter can be reset by a VUI_MBOX_ENA/VUI_MBOX_RESET control operation/argument.

RETURN VALUES
On successful completion, VUI_OK is returned. Otherwise, VUI_FAIL is returned and errno is set to indicate the error.

ERRORS
See also ioctl(2) man pages

EINVAL
The accessed mailbox has not been allocated, or it has not been allocated in buffered mode, or the ctlop/arg argument(s) is/are invalid.

4.5.16 vui_reg_base_set(), vui_reg_base_get()

SYNTAX

```
#include <sys/vme_types.h>
#include <sys/vme.h>
#include <sys/vui.h>
int vui_reg_base_set (int dev,
          vmeaddr_t regbase,
          bt_t mode)
int vui_reg_base_get (int dev,
          vmeaddr_t *regbase,
          bt_t *mode)
```

DESCRIPTION
vui_reg_base_set()
enables access to the register set of the VMEbus interface chipset from VME and sets the base address. Refer to the *Release Notes* for the required alignment and address space. Note

– that the address of the register slave window may also affect the addresses for mailboxes (e.g. FGA-5x00)

– and that, if mailboxes have been allocated, the register slave window may already be set.

`vui_reg_base_get()`
reads out the register base of the VMEbus.

VARIABLES

Variables for `vui_reg_base_set()`:

`dev`
file descriptor of an opened VME device

`regbase`
VME register base address

Variables for `vui_reg_base_get()`:

`dev`
file descriptor of an opened VME device

`*regbase`
this pointer returns the current VME register base address

RETURN VALUES

On successful completion, both functions return `VUI_OK`. Otherwise `VUI_FAIL` is returned and `errno` is set to indicate the error.

ERRORS

See also `ioctl(2)` man pages

`EINVAL`
Invalid request or argument.

`ENOTSUP`
Action not supported.

EXAMPLE

```
{
  ...
  int vmedev;
  vmeaddr_t regbase;

  if ((vmedev=open("/dev/vmectl", O_RDWR)) == -1)
  {
         perror("open"); … /* ERROR */
  }
  vui_reg_base_set( vmedev, 0xffe0 );
  vui_reg_base_get( vmedev, &regbase );
  printf("->register base address: 0x%lx\n", regbase);
  ...
  (void)close(vmedev);
  ...
}
```

2049369 – 0 January 2000

4.5.17 vui_reg_read(), vui_reg_write()

SYNTAX

```
#include <sys/vme_types.h>
#include <sys/vme.h>
#include <sys/vui.h>
/* if fga5000 or fga5100 based CPU board then include: */
#include <sys/fga5000.h>
/* if s4-based CPU-board then include: */
#include <sys/s4.h>


int vui_reg_read (
          int dev,
          ulong_t vmereg,
          ulong_t *regvalue);

int vui_reg_write (
          int dev,
          ulong_t vmereg,
          ulong_t regvalue);
```

DESCRIPTION

`vui_reg_read()`
allows reading a register of the VME interface which is specified by `vmereg`. `regvalue` points to the address to be used for storing the data read.

`vui_reg_write()`
allows writing a register of the VME interface which is specified by `vmereg`. `regvalue` is the value to be written to the register.

VARIABLES

Variables for `vui_reg_read()`:

`dev`
file descriptor of an opened VME device

`vmereg`
specifies the register to be read. The registers for the VME interface are defined in the corresponding include file (refer to the *Release Notes*), e.g. in `sys/fga5000.h`. Only use the literals defined in the include file, as they contain, among other information, the offset and size of the registers present. It is not necessary to know the absolute physical or virtual address of the register set.

When accessing register arrays, one should use the macro `VME_REGARR()` defined in `sys/vme_types.h`, which calculates the correct parameter for a certain index.

`*regvalue`
For a read access this variable contains the pointer to the address where the register content shall be stored.

Variables for `vui_reg_write()`:

`dev`
file descriptor of an opened VME device

`vmereg`
specifies the register to be written. The registers for the VME interface are defined in the corresponding include file (see *Release Notes*), e.g. in `sys/fga5000.h`. Only use the literals defined in the include file, as they contain – among other information – the offset and the size of the registers present. It is not necessary to know the absolute physical or virtual address of the register set.

When accessing register arrays, one should use the `VME_REGARR()` macro defined in `sys/vme_types.h`, which calculates the correct parameter for a certain index.

`regvalue`
For a write access this variable contains the value which shall be written to the VME register.

RETURN VALUES
On successful completion, `VUI_OK` is returned. Otherwise, `VUI_FAIL` is returned and `errno` is set to indicate the error.

ERRORS
See also `ioctl(2)` man pages

EAGAIN
At the time of execution the action cannot be done, e.g. because no resources are available.

ECANCELED
A timeout occurred.

EINVAL
Invalid request or argument, or no more ranges are available.

ENOTSUP
Action not supported.

EXAMPLE

```
#include <sys/fga5000.h>
...
int vmedev;
u_long regvalue;
if ((vmedev=open("/dev/vmectl", O_RDWR)) == -1)
{
        perror("open"); … /* ERROR */
}
vui_reg_write(vmedev, F50_REG_FMB_ADDR, 10);
vui_reg_read(vmedev, VME_REGARR(F50_REG_VME_RANGE, 6),
                &regvalue);
...
(void)close(vmedev);...
}
```

2049369 – 0 January 2000

4.5.18 vui_reset()

SYNTAX

```
#include <sys/vme_types.h>
#include <sys/vme.h>
#include <sys/vui.h>
int vui_reset(int dev)
```

DESCRIPTION

`vui_reset()`
resets all VMEbus CPU boards except for the one the driver is running on by triggering VME SYSRESET.

Note: **See the *Release Notes* whether this function is supported or not. If supported, the functionality of this call depends on the CPU board's switch setting. Enabling or disabling the SYSRESET output and input signal is switch-selectable. Therefore, check the CPU board's switch setting to ensure proper operation.**

VARIABLES

`dev`
file descriptor of an opened VME device

RETURN VALUES

On successful completion, `VUI_OK` is returned. Otherwise, `VUI_FAIL` is returned and `errno` is set to indicate the error.

ERRORS

See also `ioctl(2)` man pages

EINVAL
Invalid request or argument, or no more ranges are available.

ENOTSUP
Action not supported.

EXAMPLE

```
{ ...
  int vmedev;

  if ((vmedev=open("/dev/vmectl", O_RDWR)) == -1)
  {
          perror("open"); … /* ERROR */
  }
  vui_reset (vmedev);
  ...
  (void)close(vmedev);
  ...
}
```

4.5.19 vui_sysfail_assert(), vui_sysfail_deassert()

SYNTAX

```
#include <sys/vme_types.h>
#include <sys/vme.h>
#include <sys/vui.h>
int vui_sysfail_assert(int dev)
int vui_sysfail_deassert(int dev)
```

DESCRIPTION

`vui_sysfail_assert()`
asserts sysfail line.

`vui_sysfail_deassert()`
clears sysfail line.

VARIABLES

`dev`
file descriptor of an opened VME device

RETURN VALUES

On successful completion, both functions return `VUI_OK`. Otherwise, `VUI_FAIL` is returned and `errno` is set to indicate the error.

ERRORS

See also `ioctl(2)` man pages

EAGAIN
At the time of execution the action cannot be done, e.g. because no resources are available.

ECANCELED
A timeout occurred.

EINVAL
Invalid request or argument, or no more ranges are available.

ENOTSUP
Action not supported.

EXAMPLE

```
{
  ...
  int vmedev;

  if ((vmedev=open("/dev/vmectl", O_RDWR)) == -1)
  {
          perror("open"); … /* ERROR */
  }

  vui_sysfail_assert (vmedev);
  ...
  sleep(1);
  vui_sysfail_deassert(vmedev);
  ...
  (void)close(vmedev);
  ...
}
```

2049369 – 0 January 2000

4.5.20 vui_(n)sysfail_wait(), vui_(n)sysfail_signal()

SYNTAX

```
#include <sys/vme_types.h>
#include <sys/vme.h>
#include <sys/vui.h>
int vui_sysfail_wait (int dev)
int vui_sysfail_signal (int dev, int signal)
int vui_nsysfail_wait (int dev)
int vui_nsysfail_signal (int dev, int signal)
```

DESCRIPTION

`vui_sysfail_wait()/vui_nsysfail_wait()`
waits for the VME SYSFAIL* to be

- asserted in case of `vui_sysfail_wait()`
- or negated in case of `vui_nsysfail_wait()`.

`vui_sysfail_signal()/vui_nsysfail_signal()`
sets the signal which shall be sent to the user application when VME SYSFAIL* is

- asserted in case of `vui_sysfail_signal()`
- or negated in case of `vui_nsysfail_signal()`.

VARIABLES

Variables for `vui_(n)sysfail_wait()`:

`dev`
file descriptor of an opened VME device

Variables for `vui_(n)sysfail_signal()`:

`dev`
file descriptor of an opened VME device

`signal`
signal to be sent to the user application when VME SYSFAIL* is asserted. By default the `vmectl` driver is configured to detect only transitions of the SYSFAIL line from high to low or vice versa, but not the current state of the SYSFAIL line. This behavior can be changed by modifying the driver configuration file. For further information refer to the respective comment in `vmectl.conf`.

If set back to `0`, sending the signal is stopped.

Note: **Not all signals can be sent to an application. For a list of possible signals, see the `proc_signal(9F)` man page.**

RETURN VALUES

On successful completion, both functions return `VUI_OK`. Otherwise `VUI_FAIL` is returned and `errno` is set to indicate the error.

ERRORS

See also `ioctl(2)` man pages

EAGAIN
At the time of execution the action cannot be done, e.g. because another process is using the SYSFAIL functions.

EINVAL
Invalid request or argument.

ENOTSUP
Action not supported.

EXAMPLE 1

```
{
  int vmedev;
  if ((vmedev=open("/dev/vmectl", O_RDWR)) == -1)
  {
         perror("open"); … /* ERROR */
  }

  vui_sysfail_wait (vmedev);
  ...
  (void)close(vmedev);
  ...
}
```

EXAMPLE 2

```
{
  int vmedev;
  if ((vmedev=open("/dev/vmectl", O_RDWR)) == -1)
  {
         perror("open"); … /* ERROR */
  }
  ...
  sigset( SIGINT, acfail_hdl );
  vui_sysfail_signal(vmedev, SIGINT);
  ...
  vui_sysfail_signal(vmedev, 0);
  ...
  (void)close(vmedev);
  ...
}
```

4.5.21 vui_error_info()

SYNTAX

```
#include <sys/vme_types.h>
#include <sys/vme.h>
#include <sys/vui.h>
int vui_sysfail_wait(vme_errinfo_t *err_infop, u_int flags)
```

DESCRIPTION `vui_error_info()`
retrieves error counters from VME fault handling routines.

Note: **Errors caused by DMA transactions are not covered by this mechanism since they are handled by the DMA interface.**

2049369 – 0 January 2000

VARIABLE TYPES

`vme_errorinfo_t` is defined in `vme.h`. It is used to count various errors that have occurred during runtime.

```
typedef struct vme_errinfo
{
  u_int  vme_werrs;/* total #of write errors */
  u_int  vme_wp_errs;/* #of VME write posted errors */
  u_int  vme_rerrs;/* #of VME read errors */
  u_int  lbus_wp_errs;/* #of localbus posted write err. */
  u_int  iack_errs;/* #of IACK errors */
  u_int  res1, res2, res3;
} vme_errinfo_t;
```

`vme_werrs`
total number of posted and non-posted VMEbus write errors

`vme_wp_errs`
total number of VMEbus posted write errors

`vme_rerrs`
number of VMEbus read errors

`lbus_wp_errs`
number of posted local bus write errors that have not been reported to the accessing VMEbus master by asserting BERR

`iack_errs`
number of failed interrupt acknowledge cycles initiated by the local CPU acting as interrupt handler

VARIABLES

`dev`
file descriptor of an opened VME device

`*error_infop`
pointer to a VME error count structure.

`flags`
is a bit set which may contain the following elements:

`VME_SLEEP`	waits for the next error event increasing one of the error counters before returning counters. The wait state is interruptible by a signal. The data stored to `err_infop` will be updated even if the wait state was interrupted by a signal.
`0`	returns counters immediately.

Note: **It may be that error events are dropped when using the flag `VME_SLEEP`. This is the case when an error occurs in the time between issuing one of the above function calls and actually waiting for an error event. To prevent such problems, the application**

programmer should set a timeout which interrupts the wait state from time to time and then check the error counters.

RETURN VALUES

On successful completion VUI_OK is returned. Otherwise VUI_FAIL is returned and `errno` is set to indicate the error.

ERRORS

See also `ioctl(2)` man pages

EINTR
A wait state has been interrupted by a signal. The contents of the counter values will be up to date

EINVAL
Invalid flags have been provided or `err_infop` was `NULL`.

EXAMPLE

```
{ ...
  int vmedev;
  vme_errinfo_t err_infop;
  if ((vmedev=open("/dev/vmectl", O_RDWR)) == -1)
  {
          perror("open");
  }

  vui_error_info (vmedev, &err_infop, 0);

  printf("->total number of write errors: %d\n",
                  err_infop.vme_werrs);
  ...
  (void)close(vmedev);
  ...
}
```

2049369 – 0 January 2000

5 Device Driver Developer's Guide

Device driver developers need to know the interfaces between the nexus driver and the leaf drivers. The Force Computers VMEbus nexus driver provides 2 such interfaces:

1. the standard Solaris DDI/DDK: For information on DDI/DDK, see the man pages and the Solaris manual on writing device drivers.
2. the VDI – the VME driver interface: It is an extension of the DDI/DDK to support the VMEbus capabilities of CPU boards from Force Computers. It provides a standard interface. The VDI is described in this section.

Example

For an example on how to use the VDI functions see the source code of the leaf drivers included in the Solaris VMEbus Driver package.

5.1 VME Nexus Driver Configuration

Configuration file

`/platform/`*`arch`*`/kernel/drv/VME.conf` is the configuration file of the VMEbus nexus driver. Depending on the hardware platform, *`arch`* may be either `sun4m` or `sun4u`.
Within the configuration file values may be assigned to the master and slave window properties (see below). However, the default values are defined to be suitable for standard use of the Solaris VMEbus Driver package. Only when integrating drivers which are not included in the Solaris VMEbus Driver package (but, e.g., included in third party products) or when customizing of drivers is necessary, the values of the `VME.conf` properties may have to be changed.

5.1.1 Master Window Properties

Under normal circumstances it is not necessary to define master windows in `VME.conf` because master windows will be set up dynamically when a driver requests one.
However, this feature may help saving resources of the VMEbus interface chip because only a limited amount of master windows is available (e.g., 16 on the FGA-5000).
Master windows which are defined in `VME.conf` are always 1 slot in size, i.e. 256 MByte.

Example:

Assume that there are several VMEbus devices which occupy adjacent address ranges in the A32/D32 space. If so, it is possible to define 1 master range in `VME.conf` that covers all devices. When one of the corresponding device drivers requests a master range, the VMEbus nexus driver will notice that there already exists a range which covers the requested one. The result is, that only 1 instead of several master windows of the VMEbus hardware have to be reserved.

The remaining parts of this section

- first describe the master window properties which can be set:
  - properties for master windows (see "Master windows" below),
  - properties to use programmable AM codes in conjunction with a bus property specifying an address space (see "Programmable AM codes" below),
- followed by a description of how to specify defaults (see page 102).

Master windows

The following properties control master window allocation:
`vmewin=`*`vmeaddr[,size,bus-properties]`*
`vmewin`*`X`*`=`*`vmeaddr[,size,bus-properties]`*

`vmeaddr`
specifies the VMEbus address to be covered by the master window.

`size`
specifies the number of bytes of VMEbus address space to be mapped. If this parameter is not specified, 256MByte are allocated at `vmeaddr`.

`bus-properties`
is a set of bits which specifies the VMEbus address space and other properties of the master window (see section 6 "VME Bus Properties" on page 159).

There are CPU boards supporting more than 1 master window at a time (e.g., all CPU boards with FGA-5000 interface). For these CPU boards, use `vmewin` for the 1st and `vmewin`*`X`* for all following windows where *`X`* is replaced by a digit.

Sample definition of 3 master windows:

```
# VME address 10000000_16, size 256 MB, A32 space,
# privileged data access, max. data width is 32 bit.
#
vmewin=0x10000000, 0x10000000, 0x00040004

# VME address 20000000_16, size 256 MB, A32 space,
```

```
# privileged data access, max. data width is 16 bit,
# write posting is enabled.
#
vmewin1=0x20000000, 0x10000000, 0x01040002

# VME address 0, size 4KB, A16 space,
# privileged data access, max. data width is 32bit,
#
vmewin2=0x00000000, 0x1000, 0x00010004
```

Programmable AM codes

There are VMEbus interface chips which offer programmable AM codes (also called user-defined AM codes). The FGA-5100 for example, allows to program 2 user-defined AM codes for VME master windows and to select one of them individually for each master window. This is supported by the VMEbus nexus driver via the `pamc` property (`pamc` – programmable AM code). The 2 AM codes are fixed during system run-time, i.e. there is no driver interface to set the AM codes dynamically.

– specifying

- The 2 programmable AM codes are specified via the `pamc` property in the configuration file of the VME nexus driver, `VME.conf`.

Note: **When changing a value for a programmable AM code in `VME.conf`, the `vmeplus.conf` has to be updated as well so that the bus property for the corresponding `reg` property reflects the new value in the `VME.conf` file.**

`pamc` consists of 2 integer values defining the programmable AM codes 1 and 2. If no `pamc` property is defined in `VME.conf`, default values are used: 10_{16} for AM code 1 and 11_{16} for AM code 2.

To define the 2 codes, the following line has to be inserted into VME.conf:

`pamc=`*`value_of_AM_code_1, value_of_AM_code_2`*

Sample definition of AM code 1 = `0x15` and AM code 2 = `0x16`:

```
pamc=0x15, 0x16
```

– set up window

- To set up master windows using programmable AM codes, use the `VME_BT_PAMC1` and `VME_BT_PAMC2` bits via `vdi_map_abs()`.

Note: **The `VME_BT_PAMC1` and `VME_BT_PAMC2` bus properties do not define the address space size, which means that a `VME_BT_PAMC`*x*` bus property literal must always be used in combination with a bus property specifying an address space (`VME_BT_Axx`).**

– access devices

- Memory devices to access the VMEbus using the programmable AM codes are provided by the `vmeplus` driver:

```
/dev/vmepam1d16
/dev/vmepam2d16
/dev/vmepam1d32
/dev/vmemap2d32
```

Specifying defaults

When a device driver wants to access VMEbus locations, it usually maps them via `ddi_map_regs(9e)`. This requires a `reg` property in its configuration file *`driver`*`.conf` which specifies the correct address space. However, this might not be possible in all cases:

- Certain properties cannot be set via this interface: for example enabling or disabling write posting for a mapping.
- Others cannot be set on a per mapping basis but only globally, thereby affecting all mappings: for example generating privileged or non-privileged AM codes in case of FGA-5000 or S4 based hardware.

The default settings for such properties can be controlled by the `VME:vme_master_defaults` variable in `/etc/system`. It specifies a set of miscellaneous bus property bits as defined in `sys/vme_types.h` (see section 6.3 "Miscellaneous Bus Properties" on page 162 and the respective section of *Release Notes* for the hardware dependencies). Note that only miscellaneous bus properties can be specified here.
A device driver can override the VMEbus nexus driver's default settings by using `vdi_transfer_set()/~_get()` (see section 5.4.27 "vdi_transfer_set(), vdi_transfer_get()" on page 154). For an example, see the source code of the `vmeplus` driver, which calls `vdi_transfer_set()` at bootup to override the VMEbus nexus driver's default settings with its own defaults.

Example for FGA-5000 based hardware:

The following entry in `/etc/system`

– enables write posting for all drivers which do not override the VMEbus nexus driver's defaults via `vdi_transfer_set()` (flag `VME_BT_WP` = 0100.0000_{16})

– and provides for non-privileged access privilege for all master accesses (flag `VME_BT_NPRV` = 0800.0000_{16}).

```
set VME:vme_master_default = 0x09000000
```

2049369 – 0 January 2000

5.1.2 Slave Window Property

The slave window property is only relevant to be set correctly if a DDI compliant leaf driver wants to set up DMA transfers from a VMEbus device. By setting this property, the VMEbus nexus driver can provide a VMEbus address range where DMA capable VMEbus devices may access the DMA buffer. Ensure that the specified address range fits to the requirements of the VME DMA device(s).

Note: **If you do not use drivers performing VME DMA via the standard DDI interface, you should not define this property, because valuable hardware resources are used up by this. For the drivers included in the Solaris VMEbus Driver package there is no need to define this property.**

The following slave window property can be set if needed:
`slavewin=`*`vmeaddr, size, space`*

`vmeaddr`
defines the VMEbus start address of the slave window.

`size`
defines the size of the slave window in Bytes.

`space`
specifies the address space and the bus properties for the slave window (see section 6 "VME Bus Properties" on page 159 or see the VMEbus nexus driver's configuration file).

All 3 parameters must comply to the hardware requirements of the CPU board. Refer to the *Solaris VMEbus Driver Release Notes* and the CPU board's *Technical Reference Manual* for further information.

5.2 Device Driver Properties

A device driver is usually configured via device properties. Device properties can be specfied by means of a driver configuration file (refer to *driver*.conf(4)).
This chapter describes some extensions to standard properties which are specific to the Force Computers VME nexus driver:

- interrupt specifications (see section 5.2.1 "Non-Vectored Interrupter Handling" on page 104), and
- register specifications (see section 5.2.2 "VMEbus Mappings" on page 105).

5.2.1 Non-Vectored Interrupter Handling

Solaris differentiates between vectored and non-vectored interrupts. SBus interrupts are non-vectored (i.e. the interrupt service routine is called based on the interrupt level) whereas VMEbus interrupts are usually vectored (interrupt service routine is called based on the obtained vector).
The VME nexus provides the possibility to install a non-vectored interrupt service routine (ISR) for a device driver as well. Such an ISR is called immediately after the VMEbus interrupt has been detected by software. For VMEbus bridges that perform software IACK, this happens even before the IACK cycle has been initiated. Examples for VMEbus bridges which perform software IACK are S4 and FGA-5x00 whereas for example the Universe does not perform software IACK.
To set up such an interrupt service routine, the device driver developer has to specify `-1` as interrupt vector in the `interrupts` property of his *driver*.conf file. The VME nexus driver then reserves the given VMEbus interrupt level exclusively for this device driver, i.e. the device driver grabs this interrupt level. This has the following side effects:

- As long as a device driver has grabbed a VMEbus interrupt level, all other requests for installing an ISR for this level are rejected.
- As long as at least 1 ISR for a specific level is installed, it is not possible to grab this interrupt level by installing a handler with "vector -1".

Note: **The VME nexus driver does not attempt to perform an IACK cycle itself for interrupt levels at which such a non-vectored ISR is installed. However, hardware may require this. Therefore, the device driver developer must use `vdi_intr_acknowledge()` to obtain the interrupt vector (see section 5.4.12 "vdi_intr_acknowledge()" on page 130), even if the vector is not used.**

2049369 -- 0 January 2000

5.2.2 VMEbus Mappings

A driver that wants to map VMEbus space for accessing a device needs to declare a `reg` property in its driver configuration file (see *driver*`.conf(4)`). For VMEbus drivers, a `reg` property consists of an arbitrary number of triplets, each one describing

- the VMEbus address space and access width (the bus type),
- the start address within the selected bus type, and
- the size of the area to be mapped.

The format of the bus type is defined as follows:

- Bits `0..5` define the VMEbus AM code to be generated on the VMEbus.
- Bits `6` and `7` are evaluated for single cycle AM codes and define the maximum data width generated on the VMEbus (1 byte, 2 bytes or 4 bytes per cycle). These bits are ignored for all kinds of burst AM codes. The assignment is as shown in the following table:

Table 5 Data width encoding

Bit 7	Bit 6	Hexadecimal	Data width on VME
1	0	80_{16}	D8
0	0	00_{16}	D16 and smaller
0	1	40_{16}	D32 and smaller
1	1	$c0_{16}$	reserved

Burst Cycles with Programmed I/O

When specifying an AM code that represents a VMEbus burst operation (e.g. $0c_{16}$ for supervisory MBLT data transfers), it depends on the hardware whether burst cycles will be generated at all or what circumstances must be satisfied for doing so. Refer to the *Release Notes* for details on the hardware under consideration.
In any case, if no burst cycles are generated, accesses to VME via such a mapping will result in the corresponding single cycle AM code with a data width of D32 or smaller (for AM code $0c_{16}$ this is equivalent to a bus type of $0d_{16} + 40_{16} = 4d_{16}$).

Programmable AM codes

To make use of the programmable AM codes provided by the VME nexus, simply provide a bus type with the AM code field (bits 0..5) set to the value configured in the VME nexus' configuration file (see section 5.1 "VME Nexus Driver Configuration" on page 99). Additionally, set the data width bits as shown in table 5 "Data width encoding" on page 105 to define the maximum access width on the VMEbus.

Global properties

Some VMEbus bridges do not support to set certain properties individually per mapping. Instead, they have to be declared globally, thus affecting all VMEbus master transfers. In such a case the VME nexus driver must be configured properly before a `reg` property which reflects such a global setting becomes usable.

Example:

On FGA-5000 based boards (e.g. SPARC/CPU-5V), the selection whether to generate supervisory or non-privileged AM codes can only be made globally. The default setting is to generate supervisory accesses. When using a driver with a `reg` property denoting a non-privileged AM code, set the following line in the system configuration file `/etc/system`:

```
set VME:vme_master_default=0x08000000
```

For details on this configuration option see section 5.1 “VME Nexus Driver Configuration” on page 99.

2049369 – 0 January 2000

5.3 Nexus Driver Fault Handling

Read errors and non-posted write errors will result in a bus error signal to be sent to the originating process whenever possible.

As of release 2.1

As of Solaris VMEbus Driver release 2.1, the originating process is passed a SIGBUS signal and a signal information structure (`siginfo`) describing the exact circumstances in case of a VMEbus bus error.
For the general mechanism of how to obtain such a `siginfo` structure see the `sigaction(2)` man pages. With respect to the information provided in the manual page the `siginfo` structure describing the error is extended in the following way:

- The `si_code` element denotes the reason for the bus error signal:
  - `si_code` = `VME_BERR_W` in case of a VMEbus write error
  - `si_code` = `VME_BERR_R` in case of a VMEbus read error
- The `siginfo_t` structure pointer should be casted to a pointer to `vme_siginfo_t` which is defined in `sys/vme_types.h`. It provides the following entries:
  - `vmeaddr_t si_vmeaddr`

    This is the VMEbus address where the fault occurred.
  - `bt_t si_busprop`

    This is a set of `VME_BT_xxx` macros describing the properties of the VMEbus access (address space, data width, etc.).

For information related to read errors and error action flags, see section 3.1 "Configuration" on page 17.
Since the way the VME nexus driver can handle VMEbus errors depends on the hardware architecture, refer to the *Release Notes* for further information on the CPU board under consideration.

5.4 VDI Functions

The following table gives an overview of the functions provided by the VME driver interface:

Table 6 Overview of VDI functions

Keywords	Function(s)
Initialization	vdi_attach() (p. 111)
Master mappings	vdi_map(), ~_unmap() (p. 132) vdi_map_abs(), ~_regspec() (p. 134) vdi_reg_space() (p. 148)
Slave mappings	vdi_smem_aloc(),~_free() (p. 148) vdi_smem_map(), ~_unmap() (p. 149) vdi_smem_enable() (p. 154)
DMA controller	vdi_dma_start() (p. 115) vdi_dmac_alloc_handle() (p. 120)
VMEbus	
• arbiter	vdi_arb_mode_set(), ~_get (p. 109)
• request mode	vdi_breq_set(), ~_get (p. 113)
• request level	vdi_brl_set(), ~_get (p. 114)
• release mode	vdi_brel_set(), ~_get (p. 112)
• transfer mode	vdi_transfer_set(), ~_get (p. 154)
• interrupter	vdi_intr_acknowledge() (p. 130) vdi_intr_generate() (p. 132) vdi_virq_trigger, ~_ackwait () (p. 156)
VME events: SYSFAIL, ACFAIL, ABORT, …	vdi_event_setup(), ~_release() (p. 122)
Register access	vdi_reg_read(), ~_write() (p. 144)
Register access from VMEbus	vdi_regslave_set(), ~_get (p. 145)
Mailboxes	vdi_mbox_attach(), ~_detach() (p. 136) vdi_mbox_enable(), ~_disable() (p. 141) vdi_mbox_getinfo() (p. 141) vdi_mbox_iblock_cookie(), vdi_mbox_hilevel() (p. 143)
Read-modify-write cycles	vdi_rmw() (p. 147)
Hardware information	vdi_info() (p. 125)
Error information	vdi_error_info() (p. 121)

2049369 – 0 January 2000

5.4.1 Calling VDI functions

VDI functions can be called by any device driver if the Force Computers VME Nexus driver is loaded. Note, however, the following:

- A device driver cannot assume that the VME Nexus driver is loaded if the driver is not a child of class `vme`. It is therefore not advised to call VDI functions by device drivers that are not direct or indirect children of the `vme` class.
- When a device driver module using VDI calls is loaded, the kernel must dereference the VDI function references. This requires that a device driver declares a global variable named `_depends_on` as follows:

```
char _depends_on[]="drv/VME";
```

- To prevent loading a driver which uses VDI functions in an environment with a 3rd party VME nexus driver that does not provide VDI functionality, the driver should evaluate the return value of `vdi_attach()` in it's `probe`(9e) or `attach`(9e) routine (see section 5.4.3 "vdi_attach" on page 111.).

5.4.2 vdi_arb_mode_set(), vdi_arb_mode_get()

SYNTAX

```
#include <sys/vdi.h>
int vdi_arb_mode_set( u_int mode );
int vdi_arb_mode_get( u_int *mode );
```

DESCRIPTION

`vdi_arb_mode_set()`

controls the arbitration mode of the VMEbus arbiter. This can only be altered if the board is system controller (VMEbus slot 1).

`vdi_arb_mode_get()`

Checks whether the board is slot-1 device and returns the current arbitration mode.

Note: **Software cannot detect whether the board is mounted in VMEbus slot 1 in case of S4 based hardware. In this case, `vdi_arb_mode_get()` always returns the current setting of the arbiter, regardless whether it is enabled or not.**

VARIABLES

`mode`

arbitration mode, for possible values see below:

Table 7 Arbitration modes

VME_ARB_OFF	The board is not system controller (slot-1 device).
VME_ARB_SINGLE	Single level arbiter on level 3
VME_ARB_RR	Round robin arbiter
VME_ARB_PRI	Priority arbiter (level 3 = highest priority)
VME_ARB_PRIRR	Combined round robin / priority arbiter. Level 3 requests always have the highest priority, levels 0, 1 and 2 are handled in round robin fashion.

RETURN VALUES

VDI_SUCCESS
if successful.

VDI_INVALID
parameter mode is invalid, arbitration mode is not supported, or the CPU board is not the system controller.

2049369 – 0 January 2000

EXAMPLE

```
void ena_roundrobin()
{
  int arb;

  /* Check slot-1
   */
  vdi_arb_mode_get( &arb );

  if (arb == VME_ARB_OFF)
  {
          cmn_err( CE_WARN,
                   "Board is not mounted in slot 1");
  }
  else
  {
          /* Set round-robin arbitration
           */
          if ( vdi_arb_mode_set( VME_ARB_RR ) != VDI_SUCCESS
)
          {
                  cmn_err( CE_WARN, "Failed to set arbiter
to RR" );
          }
  }
}
```

5.4.3 vdi_attach

SYNTAX

```
#include <sys/vdi.h>
int vdi_attach( dev_info_t *dip, void *infop );
```

DESCRIPTION

`vdi_attach()` has to be called before any other VDI function in order to ensure that the driver's parent VME Nexus provides the VDI interface. The preferred location to invoke this function is within a driver's probe or attach routine.

VARIABLES

`dip`
The device info-pointer of the calling driver.

`infop`
Must be set to `NULL`.

RETURN VALUES

`VDI_SUCCESS`
if successful.

`VDI_FAILURE`
the parent VME nexus driver does not support the VDI calls. No VDI function must be called in this case.

EXAMPLE

```
static int mydrv_probe (dev_info_t *dip)
{
    if (VDI_SUCCESS == vdi_attach( dip, NULL ))
    {
        return (DDI_PROBE_SUCCESS);
    }
    /* Required VDI support not present
     */
    return DDI_PROBE_FAILURE;
}
```

5.4.4 vdi_brel_set(), vdi_brel_get()

SYNTAX

```
#include <sys/vdi.h>
int vdi_brel_set( int mode );
int vdi_brel_get( int *mode );
```

DESCRIPTION

`vdi_brel_set()`
controls how the VME bus is released after a master cycle has completed.

`vdi_brel_get()`
returns the current release mode.

VARIABLES

`mode`
bus release mode for future master transfers, for possible values see below:

Table 8 Bus release modes

`VME_BRL_ROR`	Release bus on request (BR[]* asserted)
`VME_BRL_RWD`	Release bus after transfer is completed ("Release when done")
`VME_BRL_RAT`	Release bus after timeout
`VME_BRL_ROC`	Release when bus is clear (BCLR* asserted)

RETURN VALUES

`vdi_brel_get()` returns the current release mode.

`vdi_brel_set()` returns one of the following values:

`VDI_SUCCESS`
if successful.

`VDI_INVALID`
if the parameter `mode` is invalid or the release mode is not supported.

204936 9 -- 0 January 2000

EXAMPLE

```
{
  int brel;
  ...
  if ( vdi_brel_set( VME_BRL_RWD ) != VDI_SUCCESS )
  {
          … /* ERROR */
  }
  ...
  if ( vdi_brel_get( &brel ) != VDI_SUCCESS )
  {
          … /* ERROR */
  }
  ...
}
```

5.4.5 vdi_breq_set(), vdi_breq_get()

SYNTAX

```
#include <sys/vdi.h>
int vdi_breq_set( int mode );
int vdi_breq_get( int *mode );
```

DESCRIPTION

`vdi_breq_set()`
controls how the VMEbus is requested for master transfers. The normal operation should be fair mode. Otherwise, other bus participants being further down the daisy chain may starve when many transfers are done in demand mode.

`vdi_breq_get()`
returns the current request mode.

VARIABLES

`mode`
bus request mode for future master transfers, for possible values see below:

Table 9 Bus request modes

`VME_BRQ_FAIR`	Request in fair mode, i.e. wait for BG[]* to be cleared.
`VME_BRQ_DEMAND`	Request in demand mode, i.e. assert BG[]* immediately.

RETURN VALUES

`vdi_breq_get()` returns the current request mode.

`vdi_breq_set()` returns one of the following values:

`VDI_SUCCESS`
if successful.

`VDI_INVALID`
if the parameter `mode` is invalid or the request mode is not supported

EXAMPLE

```
{
  int brm;
  ...
  if ( vdi_breq_set( VME_BRQ_DEMAND ) != VDI_SUCCESS )
  {
          … /* ERROR */
  }
  ...
  if ( (rc = vdi_breq_get( &brm )) != VDI_SUCCESS )
  {
          … /* ERROR */
  }
  ...
}
```

5.4.6 vdi_brl_set(), vdi_brl_get()

SYNTAX

```
#include <sys/vdi.h>
int vdi_brl_set( int level );
int vdi_brl_get( int *level );
```

DESCRIPTION

`vdi_brl_set()`
controls the VME request level on which the bus shall be requested for master transfers.

`vdi_brl_get()`
returns the current request level.

VARIABLES

`level`
bus request level for future master transfers (`0`, `1`, `2`, or `3`)

RETURN VALUES

`vdi_brl_get()` returns the current request level.

`vdi_brl_set()` returns one of the following values:

`VDI_SUCCESS`
if successful.

`VDI_INVALID`
if the parameter `level` is invalid or the request level is not supported.

EXAMPLE

```
{
  int brm;
  ...
  if ( vdi_brl_set( 2 ) != VDI_SUCCESS )
  {
          … /* ERROR */
  }
  ...
  if ( vdi_brl_get( &brm ) != VDI_SUCCESS )
  {
          … /* ERROR */
```

2049369 – 0 January 2000

```
  }
  ...
}
```

5.4.7 vdi_dma_start()

SYNTAX

```
#include <sys/vdi.h>
int vdi_dma_start(
          bt_t bt,
          vmeaddr_t vmeaddr,
          lbaddr_t *lbaddr,
          uint_t len,
          uint_t flags,
          int (*waitfp)(caddr_t),
          caddr_t arg,
          void (*callback)(caddr_t),
          caddr_t cbarg,
          int *rc);
```

DESCRIPTION `vdi_dma_start()` is an interface to the on-board VMEbus DMA controller. It programs the source and destination registers and starts the DMA transfer.

If the DMAC is not immediately available, the value of `(*waitfp)()` determines which action is taken. If the value of `(*waitfp)()` is `DDI_DMA_DONTWAIT`, `vdi_dma_start()` will return immediately. The value `DDI_DMA_SLEEP` will cause the thread to sleep and not return until the current DMA transfer has been finished. Any other value is assumed to be a callback function address. In that case, `vdi_dma_start()` returns immediately and the `(*waitfp)()` function is called when the DMAC might have become available (note that it will be called from a low-level interrupt context).
When the callback function `(*waitfp)()` is called, it should attempt to allocate the DMAC again. If it succeeds or does not need the DMAC any more, it must return the value `1`. If it tries to allocate the DMAC, but fails to do so, it must return `0`.
When the DMA transfer is terminated, successfully or not, the callback function `(*callback)()` is called with the argument `cbarg` from the DMA interrupt routine and `rc` is set to `VDI_SUCCESS` if it terminated successfully, otherwise to `VDI_FAILURE`. If the value of `callback` is `void (*)()1`, `vdi_dma_start()` will cause the thread to sleep and not return until the current DMA transfer has been terminated.

VARIABLES

`bt`
encoded VMEbus address modifier and access-mode (see section 6 "VME Bus Properties" on page 159).

`vmeaddr`
VMEbus address. Must fit the DMA controller's alignment constraints. They can be detected with the function vdi_info (refer to the *Release Notes* for information on hardware dependencies).

`lbaddr`
address of the memory object in the format the dma controller expects. This can be obtained by a call to `ddi_dma_buf_bind_handle(9f)`, it is the *`cookie`*`.dmac_address` value. The DMA handle required for using the DDI DMA functions can be obtained by `vdi_dmac_alloc_handle()` (p. 120).

`len`
number of bytes to transfer.

`flags`
`DDI_DMA_READ` or `DDI_DMA_WRITE`

`waitfp`
address of a function to call back later if the requested resources are not available. The function addresses `DDI_DMA_SLEEP` and `DDI_DMA_DONTWAIT` are accepted to either wait until the resources are available or not to wait (and not to schedule a callback).

`arg`
argument to be passed to the callback function `waitfp` if such a function is specified

`callback`
address of a function to call back later when the DMA transfer is terminated. Can be `NULL`.

`cbarg`
argument to be passed to the callback function `callback` if such a function is specified

`rc`
pointer to the return value of the DMA termination routine

20493G9 – 0 January 2000

RETURN VALUES

`VDI_SUCCESS`
DMA has successfully been started.

`VDI_DMA_BUSY`
DMA controller is busy. Another VME DMA transfer is currently running.

`VDI_DMA_INVALID`
unsupported bus type or invalid parameter (unaligned address, invalid size, etc.)

If the DMA transfer is terminated due to an error condition, `rc` is set to `-1`, otherwise, to `0`.

EXAMPLE

```
static void vmedma_done();
static int vmedma_rc;

static int
vmedma_strategy (register struct buf *bp)
{
    int ok = 1;
    int flags; /* flags to pass to ddi_dma_buf_setup */
    ddi_dma_cookie_t dma_cookie;
    uint_t ccount;
    int rc;

    FLOW_DPRINTF
          ((VME_DMA_DEBUG | VME_FLOW_DEBUG |
VME_LEAF_DEBUG),
          ("start vmedma_strategy()\n"));

    flags = DDI_DMA_SBUS_64BIT;

    /* Set DMA request flags based on struct buf flags */
    if (bp->b_flags & B_READ)
    {
        flags |= DDI_DMA_READ;
        DPRINTF ((VME_DMA_DEBUG | VME_LEAF_DEBUG),
                 ("vmedma_strategy(): READ\n"));
    }
    else if (bp->b_flags & B_WRITE)
    {
        flags |= DDI_DMA_WRITE;
        DPRINTF ((VME_DMA_DEBUG | VME_LEAF_DEBUG),
                 ("vmedma_strategy(): WRITE\n"));
    }
```

```
    /*
     * new DMA interface
     */
    if ((ddi_dma_buf_bind_handle (
                vmedma_handle,
                     /* previously allocated DMA handle
                     * in vmedma_attch() routine with
                     * ddi_dma_alloc_handle()
                     */
                bp,           /* pointer to buf structure
*/
                flags,         /* Action,  what to do */
                DDI_DMA_SLEEP,/* Adress of a callback
function if
                     * resources are not available now.
                     * DDI_DMA_SLEEP = wait until
                     * resources are available
                     */
                (caddr_t) 0,/* argument passed to the
callback
                     * function
                     */
                &dma_cookie,/* pointer to the first
                     * ddi_dma_cookie struct
                     */
                &ccount)/* on successfull return, count
                     * points to value representing the
                     * number of cookies for this
                     * DMA object
                     */
                != DDI_DMA_MAPPED))
  {
         cmn_err (CE_NOTE, "ERROR:
                vmedma: ddi_dma_buf_bind_handle failed");
         bp->b_resid = bp->b_bcount;
         bp->b_flags |= B_ERROR;
         bp->b_error = EIO;
         biodone (bp);
         ok = 0;
  }

if (ok)
    {
        /* uio_loffset is adjusted in physio.
         * Note that we need uio_loffset, because vmeaddr_t
         * is a 64 bit datatype!
         */
          vme_addr = (vmeaddr_t) vmedma_uiop->uio_loffset;
          DPRINTF ((VME_DMA_DEBUG | VME_LEAF_DEBUG),
                ("vmedma_strategy():
                vmeaddr (physical) = 0x%lx\n",
                (u_long) vme_addr));
```

```
                /*
                 * start DMAC
                 * if busy, sleep
                 * call vmedma_done(bp) when done.
                 */
                rc = vdi_dma_start
                        (vme_space,
                         (vmeaddr_t) vme_addr,
                         (lbaddr_t) dma_cookie.dmac_address,
                         (uint) bp->b_bcount,
                         (uint) flags,
                         DDI_DMA_SLEEP,
                         NULL,
                         vmedma_done,
                         (caddr_t) bp,
                        &vmedma_rc);

                if (rc != VDI_SUCCESS)
                {
                        DPRINTF ((VME_DMA_DEBUG | VME_ERROR_DEBUG
        |
                        VME_RESOURCE_DEBUG | VME_LEAF_DEBUG),
                         ("vmedma_strategy():
                           vdi_dma_start failed: %d\n", rc));

                        ddi_dma_unbind_handle (vmedma_handle);
                        vdi_to_errno (&rc);
                        bp->b_resid = bp->b_bcount;
                        bp->b_flags |= B_ERROR;
                        bp->b_error = rc;
                        biodone (bp);
                        ok = 0;
                }
            }

          FLOW_DPRINTF ((VME_DMA_DEBUG | VME_FLOW_DEBUG |
                        VME_LEAF_DEBUG),
                        ("end vmedma_strategy(): allways 0\n"));
          return (0);
        }
        /* end of "vmedma_strategy()" */
```

5.4.8 vdi_dmac_alloc_handle()

SYNTAX

```
#include <sys/vdi.h>
int vdi_dmac_alloc_handle(
                  dev_info_t *dip,    /* caller's dip */
                  int (*callback)(caddr_t),/* callback fct.
*/
                  caddr_t arg,        /* callback arg. */
                  ddi_dma_handle_t *handlep );/* ptr. to
handle */
```

DESCRIPTION

`vdi_dmac_alloc_handle()` allocates a DMA handle for the DMA controller built into the VMEbus interface chip. A DMA handle is required as input parameter to all other DMA related functions provided by the DDI.

`vdi_dmac_alloc_handle()` is basically identical to the DDI function `ddi_dma_alloc_handle(9f)`, except that the `attr` parameter is missing. `vdi_dmac_alloc_handle()` internally uses an attributes structure that fits to the DMA controller used within the VMEbus interface chip. Therefore, the interface to the DMA controller is hardware independent.

Except for the missing `attr` parameter, the parameters have the same semantics as described in `ddi_dma_alloc_handle(9f)`:

`dip`
is the device info-pointer of the calling device driver.

`callback`
describes the behavior if no resources are available, i.e. `DDI_DMA_SLEEP`, `DDI_DMA_DONTWAIT` or the address of a callback function.

`arg`
is the argument to be passed to the callback function described by the `callback` argument.

`handlep`
is a pointer to where the DMA handle is stored if the request is successful.

RETURN VALUES

Refer to `ddi_dma_alloc_handle(9f)`.

`VDI_NOTSUP`
the VMEbus interface chip does not have a DMA controller. In this case, the returned DMA handle is invalid.

2049369 – 0 January 2000

5.4.9 vdi_error_info()

SYNTAX

```
#include <sys/vdi.h>
int vdi_error_info( vme_errinfo_t *err_infop,
                    u_int flags);
```

DESCRIPTION `vdi_error_info()`
returns monitored amount of errors that have occurred during system runtime.

Note: **Errors caused by DMA transactions are not covered by this mechanism since they are handled by the DMA interfaces.**

VARIABLES `err_infop`
is a pointer to a structure of type `vme_errinfo_t` which is defined in `sys/vme.h` (for a description of the structure, see section 4.5.21 "vui_error_info()" on page 96)

`flags`
is a bit set which may contain the following elements:

`VME_SLEEP`	waits for the next error event increasing one of the error counters before returning counters. The wait state is interruptible by a signal. The data stored to `err_infop` will be updated even if the wait state was interrupted by a signal.
`0`	returns counters immediately.

Note: **It may be that error event(s) are dropped when using the flag `VME_SLEEP`. This is the case when an error occurs in the time between issuing one of the above function calls and actually waiting for an error event. To prevent such problems, the application programmer should set a timeout which interrupts the wait state from time to time and should then check the error counters.**

RETURN VALUES `VDI_INTR`
if a wait state has been interrupted by a signal.

`VDI_INVALID`
if invalid flags were provided.

`VDI_SUCCESS`
in all other cases.

5.4.10 vdi_event_setup(), vdi_event_release()

SYNTAX

```
#include <sys/vdi_types.h>
#include <sys/vdi.h>
int vdi_event_setup(
          dev_info_t *dip,
          int event,
          void (*cb)(caddr_t)cb,
          int cbarg );
int vdi_event_release(
          dev_info_t *dip,
          int event );
```

DESCRIPTION

`vdi_event_setup`()
installs a callback function for the specified VMEbus event for the calling device. If the event has already been attached successfully by some driver, the function fails.

`vdi_event_release`()
detaches the driver identified by `dip` from the specified event, i.e. the event handler will not be called any more and the default behavior is resumed.

VARIABLES

`dip`
device info pointer of calling device

`event`
event type to control; for possible values see below.

Note: **It is recommended to set the `IMM_CALLBACK` flag, because there is no other way to request the current status of the ACFAIL and SYSFAIL lines.**

2049369 -- 0 January 2000

Table 10 VMEbus events

Literal	Description	and default behavior
VME_SYSFAIL	VME SYSFAIL line is asserted; can be ORed with IMM_CALLBACK (see below)	see section 3 "Installation and Configuration Guide" on page 15
VME_NSYSFAIL	VME_SYSFAIL line is negated; can be ORed with IMM_CALLBACK (see below)	
VME_ACFAIL	VME ACFAIL line is asserted; can be ORed with IMM_CALLBACK (see below)	
VME_NACFAIL	VME ACFAIL line is negated; can be ORed with IMM_CALLBACK (see below)	
IMM_CALLBACK	Optional flag to be used together with one of the above literals. • If set, vdi_event_setup() checks whether the selected event is currently active. If this is the case, the callback function is scheduled immediately. • If not set, the callback function is scheduled at the next high-to-low transition (VME_SYSFAIL, VME_ACFAIL) or low-to-high transition (VME_NACFAIL, VME_NSYSFAIL), regardless of the current state.	
VME_ABORT	Abort switch on the front panel is triggered	System enters the PROM monitor. If the system has been booted with the kernel debugger, it will jump into kadb, instead.

`cb`
pointer to function to be called when the event occurs. It is scheduled in low-level interrupt context.

`cbarg`
argument to be passed to callback function

RETURN VALUES

`vdi_event_setup()` returns one of the following values:

`VDI_SUCCESS`
if the callback function is installed successfully.

`VDI_FAILURE`
if the specified event is already attached or the VME interface hardware does not support receipt of this event.

`vdi_event_release()` returns one of the following values:

`VDI_SUCCESS`
if the callback function is removed successfully.

`VDI_FAILURE`
if the event has been set up by some other device (not the one specified by `dip`) or if it hasn't been set up at all.

EXAMPLE

```
static kmutex_t event_mutex;
static kcondvar_t event_cv;

void myinit()
{
  mutex_init( &event_mutex, “mymutex”, MUTEX_DRIVER, NULL);
  cv_init( &event_cv, “mycv”, CV_DRIVER, NULL);
}

/* event callback function. Trigger conditional variable and
 * exit.
 */
static void
event_cb( void* arg )
{
  /* This may block until cv_wait is called (see below)
   */
  mutex_enter( &event_mutex );
  cv_signal( &event_cv );
  mutex_exit( &event_mutex );
}

void
wait_for_sysfail_negated()
{
  /* Wait for SYSFAIL to be cleared. The mutex is necessary
   * because our callback funtion may get called before
   * vdi_event_setup() returns (if SYSFAIL is already
   * cleared).
   */
  mutex_enter( &event_mutex );

  if ( vdi_event_setup( vmectldip,
                VME_NSYSFAIL|IMM_CALLBACK,
                event_cb, (void *)NULL) != VDI_SUCCESS )
  {
        cmn_err(CE_WARN, "vdi_event_setup failed");
  }
```

```
  else
  {
          /* wait for callback function to be triggered
           */
          cv_wait( &event_cv, &event_mutex );

          /* SYSFAIL has been cleared, release event
           */
          if ( vdi_event_release(vmectldip, VME_NSYSFAIL)
                  != VDI_SUCCESS)
          {
                  cmn_err(CE_WARN, "vdi_event_release
failed");
          }
  }
  mutex_exit( &event_mutex );
}
```

5.4.11 vdi_info()

SYNTAX

```
#include <sys/vdi.h>
void vdi_info( vdi_info_t **info );
```

DESCRIPTION

`vdi_info()` returns hardware information within the following structure of type `vdi_info_t` which is defined in `vdi_types.h`:

```
struct vdi_info
{
  int     hostbus;
          /* Bus the VME bridge resides on. */
  char    if_name[32];
          /* Name of VME interface hardware */
  char    cpu_name[32];
          /* Name of CPU board */
  int     if_rel;
          /* Version of interface hardware */
  int     lca_rel;
          /* Version of LCA */
  event_t events;
          /* Events which can be used */
  int     event_ipl;
          /* Interrupt priority event handlers */
  vdi_arb_capabilities_t
          *arb_caps;
          /* Arbiter capabilities */
  vdi_req_capabilities_t
          *req_caps;
          /* Requestor capabilities */
  vdi_dma_capabilities_t
          *dma_caps;
          /* DMA capabilities */
  vdi_event_capabilities
          *event_caps;
```

```
        /* event capabilities */
  vdi_master_capabilities_t
        *master_caps;
        /* master mapping capabilities */
  vdi_slave_capabilities_t
        *slave_caps;
        /* slave mapping capabilities */
}
typedef struct vdi_info vdi_info_t;
```

Note: **The contents of the various capability structures are mainly used internally by the VME nexus driver and not all of them are of use and interest to the device driver developer. For this reason, only a subset of their contents is listed here. Refer to `sys/vdi_types.h` for further information.**

`hostbus`
denotes the local bus the VMEbus bridge resides on. Currently it can be one of `VME_IOB_SBUS`, `VME_IOB_PCI`, or `VME_IOB_MBUS`.

`if_name`
is a string containing the name of the localBus-to-VMEbus interface chip, also called the hardware identifier, e.g. `FGA-5000` (refer to the *Release Notes*).

`cpu_name`
is a string containing the name of the CPU board. It is the value of the `name` property in the `OBP` root node.

`if_rel`
is the revision number of the VMEbus interface hardware.

`events`
specifies the events that can be used in `vdi_event_setup() / ~_release()` (p. 122).

`event_ipl`
is the interrupt level at which event handlers will be called by the VDI.

`arb_caps`
is a pointer to a structure describing the capabilities of the board's VMEbus arbiter. It is NULL if no software arbiter support is present. The structure contains the `u_int arbiter_modes` bit mask describing the arbiter modes that can be programmed (`VME_ARB_xx` literals, see `vme_types.h`).

`req_caps`
is a pointer to a structure describing the capabilities of the board's VMEbus requester. It is NULL if no software requester support is present. The structure consists of:

2049369 -- 0 January 2000

- `char br_lvls[4]`
  which lists the bus request levels that can be programmed. Each element can contain the numbers `0` through `3`, denoting a bus request level.
- `u_int br_reqmodes`
  which is a bit set describing the request modes that can be programmed (`VME_BRQ_xxx` literals, see `vme_types.h`).
- `u_int br_relmodes`
  which is a bit set describing the release modes that can be programmed (`VME_BRL_xxx` literals, see `vme_types.h`).

`dma_caps`
is a pointer to a structure describing the capabilities of the board's DMA controller. It is NULL if no DMA controller is present. The structure consists of:

- `vhi_bt_cap_t bustypes`
  which is a structure containing the VMEbus properties that can be programmed for DMA transfers (see `sys/vme_types.h`).

Note: **Note that for compatibility reasons the variable still is called `bustypes` although bus properties are meant here.**

- `vmeaddr_t vme_align`
  which is the required alignment for the VMEbus start and end address used for DMA transfers.
- `lbaddr_t buf_align`
  which is the required alignment for the DMA buffer's start and end address used for DMA transfers.

`event_caps`
is a pointer to a structure describing the capabilities of the board with respect to events (ACFAIL, SYSFAIL, ABORT, etc.). It is NULL if no event support is present. The structure consists of:

- `event_t trigger_mask`
  which is a bit mask describing the events that can be triggered (including VMEbus interrupts).
- `event_t assert_mask`
  which is a bit mask describing the events whose VME status lines can be asserted or negated (for example for ACFAIL or SYSFAIL).

`master_caps`
is a pointer to a structure describing the board's capabilities with respect to master transfers. The structure consists of:

- `int nranges`
  which is the number of available master windows.

– `u_int flags`
which contains various flags.

If `ASPACE_OVERLAP` is set, a master window provides access to all VMEbus address ranges, like for example on S4 based hardware where the A24 space is taken from the last 16 MByte of the A32 master window and the A16 space from the last 64K of the A24 range.

The `ODD_256M` flag indicates, in addition to the above, the special implementation of the S4 chip on Force Computers' CPU boards, where overlapping of address ranges (thus access to A16 and A24) is only possible if the A32 range lies on an odd 256 MByte VMEbus address boundary (refer to the *Release Notes* for details).

– `vhi_bt_cap_t bt_win`
which is a structure containing information about the properties of the master windows.

– `bt_t bt_global`
which lists the bus properties that can be applied globally to all master windows.

`slave_caps`
is a pointer to a structure describing the capabilities of the board's VMEbus slave interface. The structure consists of:

– `int nranges`
which specifies the number of available slave windows.

– `vhi_bt_cap_t bt_win`
which is a structure containing information about the bus property capabilities of the available slave windows.

– `bt_t bt_global`
which contains the bus properties that can be applied globally to all slave windows.

– `vhi_bt_cap_t bt_regslave`
which is a structure containing information about the capabilities of the slave window that provides access to the VMEbus interface registers from VME.

The structure of type `vhi_bt_cap_t` is used to describe properties of objects that can have certain bus properties (usually master or slave windows). It is necessary to distinguish between 3 possibilities:

– the bus property can be enabled or disabled by the programmer,

– the bus property is always enabled,

– or the bus property is always disabled.

2049369 – 0 January 2000

This is expressed by the structure elements `bt_change` and `bt_fix`, which are both of type `bt_t`:

State of bit #n in		
`bt_change`	**`bt_fix`**	**Implication**
set	cleared	bus prop. #n can be switched on or off
cleared	cleared	bus property #n is always switched off
cleared	set	bus property #n is always switched on

VARIABLES

`info`

pointer to `vdi_info_t` structure pointer. Do not change any contents of this structure, as it is used globally throughout the VME nexus and leaf drivers.

EXAMPLE

```
{
  vdi_info_t *misc_info;
  ...
  if ( vdi_info( &misc_info ) != VDI_SUCCESS )
  {
          … /* ERROR */
  }
  ...
}
```

5.4.12 vdi_intr_acknowledge()

SYNTAX

```
#include <sys/vdi.h>
int vdi_intr_acknowledge( dev_info_t *dip, u_int inumber);
```

DESCRIPTION

`vdi_intr_acknowledge()`
returns the obtained interrupt vector if the interrupt acknowledge cycle completed successfully.

VARIABLES

`dip`
device info pointer of calling device

`inumber`
specifies the index of the (level, vector) pair in the `interrupts` property which describes the interrupt to be acknowledged. `inumber` is zero based (see `ddi_add_intr(9F)`).

RETURN VALUES

`VDI_SUCCESS`
if successful.

`VDI_INVALID`
in case of an invalid parameter, most likely `inumber` is out of limits.

`VDI_FAILURE`
if the interrupt vector could not be obtained due to a VME bus error or a timeout.

2049369 -- 0 January 2000

EXAMPLE

Fragment of *mydriver*.conf:

```
# grab VME level 1 by specifying a vector of -1
#
interrupts=1,-1

# Device registers at VME A32D32 @0x40000000, size 4K
#
reg=0x4d,0x40000000,0x1000
```

Fragment of driver source:

```
#include <sys/types.h>
#include <sys/sunddi.h>
#include <sys/vdi.h>
...
static char *regs;
static dev_info_t *mydip;
static u_int my_intr( caddr_t arg );
static int
mydriver_attach( dev_info_t *dip, ddi_attach_cmd_t cmd )
{
  mydip = dip;
  (...)

  /* Install an interrupt handler "my_intr" for interrupt
   * property 0. Pass the property number as argument
   */
  if (ddi_add_intr( dip, 0, NULL, NULL, my_intr, 0 )
          != DDI_SUCCESS)
  {
          cmn_err( CE_WARN, "failed to grab interrupt" );
          return DDI_FAILURE;
  }

  /* Map the device registers */
  (char*)ddi_map_regs( dip, 0, &regs, 0, 0 );
  (...)
}

static u_int my_intr( caddr_t arg )
{
  int vec;

  /* Do the IACK cycle and fetch the interrupt vector.
   * "arg" is the inumber of the interrupt property.
   */
  vec = vdi_intr_acknowledge( mydip, (u_int)arg );
  if (vec < 0)
  {
          cmn_err( CE_WARN, "my_intr: IACK failed: %d", vec
);
  }
  ...
  /* Access the reg. to clear the interrupt (example) */
  regs[0] = 0xff;
  return DDI_INTR_CLAIMED;
}
```

5.4.13 vdi_intr_generate()

SYNTAX

```
#include <sys/vdi.h>
int vdi_intr_generate( int level, int vector );
```

DESCRIPTION

`vdi_intr_generate()`
triggers a VMEbus interrupt. It does not provide the possibility to set the IACK time-out. For this reason, it will wait endlessly until the IACK is finished. The wait status is interruptible by a signal.

See `vdi_virq_trigger()` (p. 156) for a more flexible interface.

Note: **The generation of interrupts is hardware dependent. Therefore, refer to the *Release Notes* whether this feature is supported on the CPU board under consideration.**

VARIABLES

`level`
VME interrupt level to trigger

`vector`
interrupt vector to use

RETURN VALUES

`VDI_SUCCESS`
if successful.

`VDI_FAILURE`
if parameters are invalid or if action is not supported.

`VDI_INTR`
if the wait state has been interrupted by a signal.

5.4.14 vdi_map(), vdi_unmap()

Note: **As of Solaris VMEbus Driver release 2.1 the `vdi_map_abs()` function is supported. It is strongly recommended to use `vdi_map_abs()`, instead of `vdi_map()`.**

SYNTAX

```
#include <sys/vdi.h>
u_int vdi_map(bt_t bt, vmeaddr_t vmeaddr, u_int len)
void vdi_unmap(u_int pfn)
```

2049369 – 0 January 2000

DESCRIPTION

`vdi_map()`
allocates local bus addresses and sets up a local- to VMEbus bus mapping. If the required address range fits into an existing mapping, only a reference count for that mapping is incremented.

This routine does not set up a mapping to actually access the VME memory. It is intended to be used by drivers that want to provide a `mmap()` entry for applications. The driver's `mmap()` routine has to provide the correct page frame number when called, which can be obtained by `vdi_map()`.

`vdi_unmap()`
frees local bus addresses and decrements reference count. If the reference count reaches `0`, the corresponding entries in the VME MMU are invalidated.

VARIABLES

`bt`
bus properties as defined in `vme_types.h` (see section 6 "VME Bus Properties" on page 159): e.g. `VME_BT_A32`, `VME_BT_A24`, `VME_BT_D8`, or `VME_BT_D32`.

`vmeaddr`
absolute VMEbus address to be mapped

`len`
amount of VMEbus space to be mapped

`pfn`
page frame number, return value of `vdi_map()`

RETURN VALUES

`pfn`
page frame number of local bus address

`-1`
There is no local bus address space available or no entries in the VME MMU are available.

EXAMPLE

```
{
  u_int pfn;
  int retval = 0;
  ...
  pfn = vdi_map( vmeplusdip, VME_BT_A32 | VME_BT_D32,
                 (vmeaddr_t)0x60000000, (u_int)0x100);
  if (pfn == (u_int)-1)
  {
         cmn_err("vmeplus_segmap: vdi_map failed\n"));
         retval = EINVAL;
  }
  else
  {
         ...
         vdi_unmap(pfn);
  }
  ...
  return (retval);
}
```

5.4.15 vdi_map_abs(), vdi_map_regspec()

SYNTAX

```
#include <sys/vdi.h>
int       vdi_map_abs( dev_info_t * dip,
                  bt_t bt,
                  vmeaddr_t vmeaddr,
                  off_t len,
                  u_int flags,
                  u_int *pfnp );
int       vdi_map_regspec(
                  dev_info_t * dip,
                  int rnumber,
                  vmeaddr_t off,
                  off_t len,
                  u_int flags,
                  u_int *pfnp );
```

DESCRIPTION

`vdi_map_abs()`
is an extension to `vdi_map()` allowing extended configuration of the mapping's setup and providing error information.

`vdi_map_regspec()`
is an extension to `ddi_map_regs()` which allows mapping in VMEbus space based on a driver's `regspec` definition (`reg` property) without mapping the memory in the kernel space.

Both functions allocate local bus addresses and set up a local- to VMEbus mapping. If the required address range fits into an existing mapping, only a reference count for that mapping is incremented.
The functions do not set up a mapping to actually access the VME memory.

2049369 – 0 January 2000

They are intended to be used by drivers that want to provide an `mmap()` entry for applications. The driver's `mmap()` routine has to provide the correct page frame number when called, which can be obtained by `vdi_map()`.
The functions may be used to ensure that later mapping requests using `ddi_map_regs`() do not fail due to temporary lack of resources (e.g. VME master window). This might happen because the local-to-VMEbus mappings are not static, i.e. they are set up on demand.

VARIABLES

`dip`
device info-pointer of the calling driver.

`bt`
bus properties as defined in `vme_types.h` (see section 6 "VME Bus Properties" on page 159): e.g. `VME_BT_A32`, `VME_BT_A24`, `VME_BT_D8`, or `VME_BT_D32`.

`vmeaddr`
absolute VMEbus address to be mapped

`len`
length to be mapped

`off`
offset into the register space defined by the register set number `rnumber` (see `ddi_map_regs(9f)` man pages)

`len`
amount of VMEbus space to be mapped

`flags`
various flags controlling how the mapping is set up:

- `VDI_MAPWAIT`
  If this flag is set and there is currently no master window available, `vdi_map_flags()` waits until a master window is available. The wait status is not interruptible unless explicitly requested (see below).
- `VDI_INTERRUPTIBLE`
  The flag only has an effect if `VDI_MAPWAIT` is also set. If so, the wait status is interruptible by a signal.

`pfnp`
physical page number where the VMEbus space has been mapped to if the operation was successful

RETURN VALUES

`VDI_SUCCESS`
the request succeeded.

`VDI_INVALID`
invalid argument

`VDI_NOSPACE`
not enough space available on the host bus to map the requested VMEbus area into or not enough resources on the VMEbus interface available to fulfill the mapping request.

`VDI_INTR`
the `VDI_MAPWAIT` and `VDI_INTERRUPTIBLE` flags were set and the wait status was interrupted by a signal.

`VDI_CONFLICT`
the mapping request could not be satisfied because it conflicted with an existing mapping.

`VDI_NOTSUP`
the mapping request is not supported by the VMEbus interface chip.

`VDI_FAILURE`
the request failed for other reasons.

5.4.16 vdi_mbox_attach(), vdi_mbox_detach()

SYNTAX

```
#include <sys/vdi.h>
int vdi_mbox_attach( vdi_mbox_req_t *mboxreqp)
int vdi_mbox_detach(int mboxnum)
```

DESCRIPTION

`vdi_mbox_attach()`
attaches and acquires a mailbox. It will program the mailbox registers in the VMEbus interface chip, but will neither enable the access to it nor the IRQ. The `vdi_mbox_req` structure contains all information necessary to set up the mailbox (see below).

`vdi_mbox_attach()` returns a mailbox number which must be used as parameter for the other mailbox routines.

Note: **Note that the mailbox interrupt is not active after this call yet. Enable it by calling `vdi_mbox_enable()`.**

`vdi_mbox_detach()`
Removes a mailbox which has been allocated by `vdi_mbox_attach()`. It disables the access to the mailbox and removes the mailbox interrupt handler.

2048369 – 0 January 2000

VARIABLES

`mboxreqp`
contains all information necessary to set up the mailbox. The structure `vdi_mbox_req_t` is defined in `vdi_types.h`:

```
typedef struct vdi_mbox_req
(
        bt_t  mbox_bt;
              /* In/Out: encoded address modifier(s) */
        ulong_tmbox_offset_def;
              /* Out: assigned mailbox address */
        ulong_tmbox_offset_min;
              /* In: lower addr. of mailbox addr. range */
        ulong_tmbox_offset_max;
              /* In: incl. upper addr. */
        uint_tmbox_access;
              /* In/Out: desired/actual access modes */
        uint_tmbox_irq;
              /* In: requested IRQ priorities */
        int   (*mbox_handler)(caddr_t);
              /* In: pointer to mbox IRQ handler */
        caddr_tmbox_handler_arg;
              /* In/Out: arg passed to the handler */
} vdi_mbox_req_t;
```

`mbox_bt`
bit field with each bit representing a desired address mode and data size. The #define statements for the bits are declared in `sys/vme_types.h` (e.g.: `VME_BT_A24`, `VME_BT_NPRV`, etc.; see section 6 "VME Bus Properties" on page 159). When returning, additional bits might be set, e.g., `VME_BT_NPRV` when the CPU board only supports supervisory and non-privileged accesses.

`mbox_offset_def`
returns the VMEbus address of the allocated mailbox if `vdi_mbox_attach()` succeeds.

`mbox_offset_min` and `mbox_offset_max`
specifies the address range in which the mailbox shall be allocated.

`mbox_access`
determines by what kind of access the mailbox is triggered. The possible values are defined in `sys/vme_types.h`:

- `VME_MB_RD` if the mailbox shall be triggered by a read access,
- `VME_MB_WR` if the mailbox shall be triggered by a write access,
- or `VME_MB_RDWR` for both read and write access.

If this parameter is set to `0`, it is implicitly set to a value supported by the CPU board.

`mbox_irq`
is a set of preferred interrupt priorities. A mailbox may interrupt on 7 interrupt levels, which are equivalent to the 7 VMEbus interrupt levels. The VMEbus interrupt levels are mapped to the respective processor interrupt levels according to the SPARC architecture. For example, a mailbox interrupt handler at level 7 runs at the same processor interrupt level as a VMEbus interrupt service routine for level 7.

The `mbox_irq` parameter is a bit set of `VME_MBOXIRQ1`,..., `VME_MBOXIRQ7` literals. If several literals are specified, `vdi_mbox_attach`() selects the lowest level supported by hardware. Setting this to `0` is equivalent to suggesting all supported interrupt levels.

Upon successful return, `mbox_irq` reflects the actual interrupt level selected. For further information on interrupt priorities, see "vdi_mbox_iblock_cookie(), vdi_mbox_hilevel()" on page 143.

`mbox_handler`
is called on receipt of the appropriate mailbox interrupt. If `mbox_handler` is set to `NULL`, no interrupt handler will be installed and no registers will be set up; it is assumed to be an advisory call, then.

`mbox_handler_arg`
argument of the `mbox_handler` routine. If `0` is passed as argument, the callback function will be called with the selected mailbox index as argument.

RETURN VALUES

`vdi_mbox_attach()` returns one of the following values:

if successful a value equal or greater `0` is returned. This is the identifier needed for referencing the allocated mailbox.

A value less than `0` indicates an error.

`VDI_BUSY`
a mailbox has been found which fits the given properties, but it is already allocated.

`VDI_INVRANGE`
the offset range in the `vme_mbox_req` structure is invalid.

`VDI_INVBT`
a requested bus property flag is not applicable.

`VDI_INVACC`
the access-mode is invalid.

2048369 – 0 January 2000

`VDI_FAILURE`
one or more arguments are invalid. Possible reasons are that no mailbox could be found which fits the given properties.

`VDI_NORESOURCES`
no mailbox is available.

`vdi_mbox_detach()` returns one of the following values:

`VDI_SUCCESS`
successful.

`VDI_FAILURE`
the mailbox number is invalid.

`VHI_INVALID`
the index is out of range.

EXAMPLE

```
vdi_mbox_req_t mbox_req;
int mbox_num;
ddi_iblock_cookie_t mbox_cookie;
kmutex_t mbox_mutex;

/* Fill the mailbox request structure */
mbox_req.mbox_bt = VME_BT_D8|VME_BT_A16;
mbox_req.mbox_offset_min = (vmeaddr_t)0;
mbox_req.mbox_offset_max = (vmeaddr_t)0xffff;

/* We don't care about the access method and let
* the VDI choose it. */
mbox_req.mbox_access = 0;

/* Don't care for a specific irq level. The VDI
will
* choose the one with the lowest priority. */
mbox_req.mbox_irq = 0;

/* Specify the callback function. We pass zero as
* argument to indicate that the interrupt handler
* should pass us the mailbox id as argument */
mbox_req.mbox_handler = mbox_intr;
mbox_req.mbox_handler_arg = NULL;

if ((mbox_num = vdi_mbox_attach (&mbox_req)) < 0)
{
        cmn_err(CE_WARN, "Failed to attach to
mailbox");
        return;
}

/* Our sample application is not capable of
handling
* hi-level interrupts */
if (vdi_mbox_hilevel(mbox_num)) {
        cmn_err(CE_WARN, "hi-level interrupt not
sup.");
        return;
}

/* initialize mutex for callback function */
vdi_mbox_iblock_cookie( mbox_num, &mbox_cookie );

mutex_init( &mbox_mutex, "my mbox mutex",
                MUTEX_DRIVER, (void*)mbox_cookie
);

/* now enable the Mailbox
  */
 vdi_mbox_enable( mbox_num );

/* Our callback function mbox_intr() is now
receiving
 * mailbox interrupts */
```

2049369 – 0 January 2000

5.4.17 vdi_mbox_enable(), vdi_mbox_disable()

SYNTAX
```
#include <sys/vdi.h>
int vdi_mbox_enable( int mboxnum)
int vdi_mbox_disable( int mboxnum)
```

DESCRIPTION

`vdi_mbox_enable()`
enables the access to the mailbox address and the mailbox IRQ. This can be used to enable the mailbox after a call to `vme_mbox_attach()` or `vme_mbox_disable()`.

`vdi_mbox_disable()`
disables the mailbox IRQ and the access to it.

VARIABLES

`mboxnum`
mailbox ID

RETURN VALUES

`VDI_SUCCESS`
if successful.

`VDI_FAILURE`
if the mailbox number is invalid.

`VDI_BUSY`
if the specified index refers to an item which has already been allocated.

`VDI_INVALID`
if at least one of the parameters does not meet the restrictions in the capability structure.

EXAMPLE
see "vdi_mbox_attach(), vdi_mbox_detach()" on page 136

5.4.18 vdi_mbox_getinfo()

SYNTAX
```
#include <sys/vdi.h>
int vdi_mbox_getinfo( vdi_mbox_info_t *mboxinfop)
```

DESCRIPTION

`vdi_mbox_getinfo()`
fills the mboxinfo structure with the information about the actual available mailboxes. The `vdi_mbox_info` structure is maintained in the VDI layer. It is updated with every `vdi_mbox_attach()` and `vdi_mbox_detach()`.

VARIABLES

`mboxinfop`
pointer to a structure containing all available information about mailboxes. The `vdi_mbox_info` structure provides information about available mailboxes and is defined in `vdi_types.h`:

```
tyepdef struct vdi_mbox_info
{
  int     nmbox;/* number of available mboxes */
  int     nmbox_inuse;/* number of attached mboxes */
  bt_t    mbox_bt;/* encoded addr. modifier(s) */
  ulong_t mbox_offset_def;/* default offset */
  ulong_t mbox_offset_mask;/* changeable bits in default */
  uint_t  mbox_access;/* possible access modes */
  irq_t   mbox_irq;/* possible interrupt levels */
} vdi_mbox_info_t;
```

`nmbox`
number of available mailboxes

`nmbox_inuse`
number of mailboxes which are currently attached

`mbox_bt`
informs about the maximum available address modes and data sizes (see section 6 "VME Bus Properties" on page 159).

`mbox_offset_def`
offset which might be introduced by the CPU board's register layout (see below).

`mbox_offset_mask`
indicates which bits of a mailbox address can be requested (see below).

`mbox_access`
access modes supported by the CPU board: `RD` or `WR`

`mbox_irq`
bit field indicating the possible mailbox interrupt levels (see section 5.4.16 "vdi_mbox_attach(), vdi_mbox_detach()" on page 136).

`mbox_offset_def` and `mbox_offset_mask`
describe which address can be requested for the next available mailbox. For example, if `mbox_offset_def` is `0x120` and `mbox_offset_mask` is `0xfe00`, selectable addresses are `0x120`, `0x320`, `0x520`, and so on.

RETURN VALUES

Always returns `VDI_SUCCESS`.

EXAMPLE

```
{
  vdi_mbox_info_t mbox_info;
  ...
  if ( vdi_mbox_getinfo( &mbox_info ) != VDI_SUCCESS )
  {
          … /* ERROR */
  }
  ...
}
```

5.4.19 vdi_mbox_iblock_cookie(), vdi_mbox_hilevel()

SYNTAX

```
#include <sys/vdi.h>
#include <sys/sunddi.h>
int vdi_mbox_iblock_cookie( int mboxnum,
                  ddi_iblock_cookie_t *cookiep );
int vdi_mbox_hilevel( int mboxnum );
```

DESCRIPTION

`vdi_mbox_iblock_cookie()`
initializes an iblock cookie for the given mailbox. The iblock cookie can be used for setting up a mutex which is safe to use within the call-back function of the mailbox.

`vdi_mbox_hilevel()`
returns information indicating whether the callback function of the given mailbox runs in high-level interrupt context or not.

These functions are equivalent to `ddi_get_iblock_cookie(9f)` and `ddi_intr_hilevel(9f)` respectively.

VARIABLES

`mboxnum`
the mailbox ID obtained by `vdi_mbox_attach()`.

`cookiep`
a pointer to an iblock cookie to be initialized.

RETURN VALUES

`vdi_mbox_iblock_cookie()` returns one of the following values:

`VDI_SUCCESS`
if the iblock cookie was initialized successfully.

`VDI_INVALID`
if invalid parameters were specified, e.g. if `mboxnum` does not denote an existing mailbox identifier.

`vdi_mbox_hilevel` returns `0` if the callback function for the given mailbox runs in low-level interrupt context, or `1` if it runs in high-level interrupt context.

EXAMPLE

see "vdi_mbox_attach(), vdi_mbox_detach()" on page 136

5.4.20 vdi_reg_read(), vdi_reg_write()

SYNTAX

```
#include <sys/vdi.h>
int vdi_reg_read( u_long   reg, u_long *value )
int vdi_reg_write( u_long  reg, u_long value )
```

DESCRIPTION

`vdi_reg_read`()
reads the contents of the VMEbus hardware register set specified by `reg` and stores it in `value`.

`vdi_reg_write`()
writes the contents specified by `value` to the VMEbus hardware register set specified by `reg`.

VARIABLES

`reg`
register identifier. For a list of the available register identifiers see the respective interface's header file (e.g., `fga5000.h`). For register arrays, the macro `VME_REGARR` can be used to calculate the correct parameter for a given index.

`value`
register content to be written

RETURN VALUES

`VDI_SUCCESS`
if successful.

`VDI_FAILURE`
if an error occurred while accessing the register.

`VDI_ALIGN`
`VDI_OFFSET`
`VDI_SIZE`
if the register alignment, offset, or size denoted by the `reg` parameter is invalid.

`VDI_INVALID`
if some other argument is invalid.

2049369 -- 0 January 2000

EXAMPLE

```
{
  int rc, retval = 0;
  u_long regaddr;
  u_long regval=0;

  rc = vdi_reg_write( F50_REG_FMB_ADDR, regval );
  if (rc != VDI_SUCCESS)
  {
         retval = rc;
  }

  rc = vdi_reg_read( F50_REG_FMB_ADDR, &regval );
  if (rc != VDI_SUCCESS)
  {
         retval = rc;
  }

  rc = vdi_reg_read( VME_REGARR(F50_REG_SBUS_RANGE, 2),
                          &regval );
  if (rc != VDI_SUCCESS)
  {
         retval = rc;
  }

  return(retval);
}
```

5.4.21 vdi_regslave_set(), vdi_regslave_get()

SYNTAX

```
#include <sys/vdi.h>
int vdi_regslave_set( bt_t bustype, vmeaddr_t vstart )
int vdi_regslave_get( bt_t *bustype, vmeaddr_t *vstart )
```

DESCRIPTION

`vdi_regslave_set()`

enables register access to the VME interface chip from the VMEbus and sets the base address. Note that there might be hardware specific side effects (e.g. concerning the FGA-5000: setting the register slave base address also affects the possible addresses for mailboxes).

`vdi_regslave_get()`

returns the current status of the register slave window (i.e. whether it is set or not and to which address it is set).

VARIABLES

`bustype`

bus properties of register slave (see section 6 "VME Bus Properties" on page 159)

Note: **Note that for compatibility reasons the variable still is called `bustype` although bus properties are meant, here.**

`vstart`
base address for slave registers

RETURN VALUES

`vdi_regslave_set()` returns one of the following values:

`VDI_BUSY`
if the mailbox is in use.

`VDI_INVALID`
if the parameters do not meet the requirements given in the slave capability structure.

`VDI_SUCCESS`
if successful.

`vdi_regslave_get()` returns one of the following values:

`VDI_RSWSET`
if the register slave window is currently enabled.

`VDI_RSWNOTSET`
if the register slave window is currently not enabled.

EXAMPLE

```
{
  int rc, retval;
  vmeaddr_t vstart=0;
  bt_t bt=VME_BT_A16;

  rc = vdi_regslave_set( bt, vstart);
  if (rc != VDI_SUCCESS)
  {
         /* error */
         retval = rc;
  }

  rc = vdi_regslave_get( &vstart);
  if (rc != VDI_RSWSET)
  {
         /* error */
         retval = rc;
  }

  return(retval);
}
```

2049369 -- 0 January 2000

5.4.22 vdi_rmw()

SYNTAX

```
#include <sys/vdi.h>
int vdi_rmw( caddr_t kva, u_char *data );
```

DESCRIPTION

`vdi_rmw()`
performs an atomic read-modify-write cycle on the specified address. It is assumed that `kva` represents a properly mapped VME master window (e.g. via `ddi_map_regs(9F)`).

VARIABLES

`kva`
kernel address where to perform a read-modify-write cycle

`data`
address used as data source and destination

RETURN VALUES

`VDI_SUCCESS`
if successful.

`VDI_INVALID`
if no master window is defined which covers the specified address.

`VDI_FAILURE`
if an error occurred, e.g a VME bus error.

EXAMPLE

```
{
  int rc;
  caddr_t reg;
  int minor = getminor (dev);
  u_char rmwval = 0xff;

  /* Map the address we want to RMW */
  if (ddi_map_regs (mydip, minor, &reg,
                    (off_t) vmeaddr,
                   (off_t) 0x1000 )
          != DDI_SUCCESS)
  {
          /* error */
          return ERROR;
  }
  if ((rc = vdi_rmw (reg, &rmwval)) != VDI_SUCCESS)
  {
          /* VME bus error ? */
          cmn_err( CE_WARN, "RMW at address 0x%x failed\n",
                   vmeaddr );
  }
  else
  {
          /* eval rmwval ... */
  }
  ddi_unmap_regs (vmeplusdip, minor, &reg, vmeaddr,
                      (off_t) 0x1000);
}
```

5.4.23 vdi_reg_space

```
#include <sys/vme_types.h>
#include <sys/vdi.h>
bt_t vdi_reg_space( dev_info_t *dip, int rnumber );
```

DESCRIPTION `vdi_reg_space` converts the first field (the bus type) of an entry in the driver's `reg` property into the corresponding set of bus property bits. It also applies the settings done by a `vdi_transfer_set()` call for this driver.

VARIABLES

`dip`
the device-info pointer of the calling driver.

`rnumber`
the offset into the driver's `reg` property. Refer also to `ddi_map_regs(9f)`.

RETURN VALUES Upon success, a set of `VME_BT_xxx` bus properties is returned which reflects the properties of the register specification indexed by `rnumber`. If `rnumber` is invalid, or the VMEbus access properties it reflects are not supported by the underlying hardware, the return code is zero.

5.4.24 vdi_smem_alloc(), vdi_smem_free()

SYNTAX

```
#include <sys/vdi.h>

int vdi_smem_alloc(
          uint_t length,
          vdi_smem_handle_t **handlep);

int vdi_smem_free(
          vdi_smem_handle_t *handlep);
```

DESCRIPTION

`vdi_smem_alloc()`
allocates shared memory for the VMEbus. `vdi_smem_handle` contains information necessary for `vdi_smem_map`() and `vdi_smem_free`().

Note: **Depending on the hardware architecture, shared memory might be allocated non-cached. Once non-cached memory has been allocated by `vdi_smem_alloc()`, it may no longer be available for normal use by the virtual memory system. This is because Solaris removes memory from the free list once it has been set to non-cached. However, the memory will be re-used for future slave memory requests.**

2048369 – 0 January 2000

`vdi_smem_free()`
frees memory which has been allocated by `vdi_smem_alloc()` previously.

VARIABLES `length`
length of the desired allocation in byte.

`handlep`
pointer to a pointer to a `SMEM` handle to be allocated and filled in.

RETURN VALUES `VDI_SUCCESS`
memory successfully allocated

`VDI_FAILURE`
allocation failed

EXAMPLE
```
{
  vdi_smem_handle_t *handlept;
  ...
  if (vdi_smem_alloc(0x100, &handlept))
  {
          cmn_err(CE_WARN,
                  "cannot allocate space for len %d",
                  (int)0x100);
          retval = -1;
  }
  else
  {
          ...
          vdi_smem_free(handlept);
  }
  ...
  return (retval);
}
```

5.4.25 vdi_smem_map(), vdi_smem_unmap()

SYNTAX
```
#include <sys/vdi.h>
int vdi_smem_map(
          vdi_smem_req_t *smemreqp,
          vdi_smem_lim_t *smemlimp,
          vdi_smem_handle_t *handlep);

int vdi_smem_unmap(
          vdi_smem_handle_t *handlep);
```

DESCRIPTION `vdi_smem_map()`
maps shared on-board memory to VMEbus. `vdi_smem_map()` followed by a `vdi_smem_enable()` call (p. 154) makes a pre-allocated region of DVMA memory accessible from the VMEbus. The caller

supplies a requested VMEbus address range in the SMEM request structure. The VMEbus window is then set up so that it encloses the DVMA range. The limit structure describes the limitations of the VMEbus master or requester.

`vdi_smem_unmap()`

unmaps a shared on-board memory to VMEbus mapping and disables access to the shared on-board memory. `vdi_smem_unmap()` may be called after `vdi_smem_map()`.

VARIABLES `smemreqp`

pointer to the shared memory request structure. The `vdi_smem_req` structure is defined in `sys/vdi_types.h`:

```
typedef struct vdi_smem_req
{
        bt_t    smemr_bt;
        vmeaddr_tsmemr_offset;
        uint_t  smemr_size;
        uint_t  smemr_flags;
}
vdi_smem_req_t;
```

Table 11 `vdi_smem_req` struct members

`smemr_bt`	Encoded bus capabilities (see section 6 "VME Bus Properties" on page 159)
`smemr_offset`	Desired VMEbus address

Table 11 **`vdi_smem_req` struct members (cont.)**

`smemr_size`	Shared memory size

`smemr_flags`	Information for mapping routines. There are 3 flags defined: `SMEM_FIXED`, `SMEM_PADDR`, and `SMEM_VADDR`. With driver Version 2.0.x only `SMEM_VADDR` is supported and must be set. With driver Version 2.1 the flags are defined as follows:
	`SMEM_PADDR`
	reserved for future extensions.
	`SMEM_VADDR`
	If this flag is set, the standard method of setting up the shared memory buffer is used.
	Due to hardware limitations, the VMEbus address to which the shared memory is actually mapped might differ from the requested one. Refer to the *Release Notes* for information on address offsets which are to be expected for the hardware under consideration.
	Currently this flag must be set. It may be combined with the flags described below.
	`SMEM_FIXED`
	If this flag is set, the VMEbus nexus driver sets up the shared memory at exactly the requested VMEbus address, provided that the requested VMEbus address is aligned to page boundary.
	The decoded VMEbus address range might be larger than the shared memory address range.
	Using this flag might fragment system resources more than not using the flag.
	See the *Release Notes* whether this flag is supported for the CPU board under consideration.

`smemlimp`
pointer to the shared memory limit structure. The structure `vdi_smem_lim` is defined in `vdi_types.h`. Driver versions 2.0.x ignore this structure. With driver version 2.1 this structure is used to support the specification of memory addresses via `SMEM_FIXED` (see table 11 "vdi_smem_req struct members" on page 150).

```
typedef struct vdi_smem_lim
{
        ulong_t slim_smem_lo;
        ulong_t slim_smem_hi;
        ulong_t slim_vme_lo;
        ulong_t slim_vme_hi;
        uint_t  slim_vme_size;
}
vdi_smem_lim_t;
```

Table 12 `vdi_smem_lim` struct members

`slim_smem_lo`	Low range of mapped shared memory
`slim_smem_hi`	Upper inclusive bound
`slim_vme_lo`	Low range of decoded VMEbus range
`slim_vme_hi`	Upper inclusive bound
`slim_vme_size`	Maximum size of decoded range

`handlep`
pointer to a pointer to a `SMEM` handle to be allocated and filled in.

RETURN VALUES

`VDI_SUCCESS`
memory was mapped successfully.

`VDI_FAILURE`
handle contains invalid values, or the VME interface hardware is not capable to cover the requested address range.

`VDI_NOSPACE`
the resources which are required to generate the mapping are not available.

`VDI_CONFLICT`
another window with the same bus properties exists, whose address range overlaps with the one needed to fulfil the actual request.

2049369 – 0 January 2000

EXAMPLE

```
/* slave memory limitations */
static vdi_smem_lim_t smem_lim =
{
   (ulong_t) 0x00000000,   /* Low range of mapped smem */
   (ulong_t) -1, /* High Limit, upper inclusive bound */
   (ulong_t) 0x00000000, /* Low range decoded VME range */
   (ulong_t) -1, /* High limit, upper inclusive bound */
   (uint_t) -1 /* Max size of decoded range */
};
...

{
  vdi_smem_handle_t *handlept;
  vdi_smem_req_t smemreq;
  ...
  /* Allocate some memory for the slave window */
  if (vdi_smem_alloc(0x100, &handlept))
  {
         cmn_err(CE_WARN,
                 "cannot allocate space for len %d",
                 (int)0x100);
         return ERROR;
  }

  /* Fill request struct */
  smemreq.smemr_bt = VME_BT_A32 | VME_BT_D32;
  smemreq.smemr_offset = 0x60000000;
  smemreq.smemr_size = 0x100;
  smemreq.smemr_flags = SMEM_VADDR;

  /* Map and enable the slave memory */
  if ( vdi_smem_map(&smemreq, &smem_lim, handlept))
  {
         /* error */
         vdi_smem_free(handlept);
         return ERROR;
  }
  if (vdi_smem_enable(handlept))
  {
         /* error */
         vdi_smem_unmap(handlept);
         vdi_smem_free(handlept);
         return ERROR;
  }

  ...

  /* Remove the slave memory */
  vdi_smem_unmap(handlept);
  vdi_smem_free(handlept);
  }
  ...
  return OK;
```

5.4.26 vdi_smem_enable()

SYNTAX
```
#include <sys/vdi.h>
int vdi_smem_enable(vdi_smem_handle_t *handlep);
```

DESCRIPTION `vdi_smem_enable()`
enables access to the shared on-board memory. May only be called after `vdi_smem_map()`.

VARIABLES `handlep`
pointer to a `SMEM` handle.

RETURN VALUES `VDI_SUCCESS`
memory successfully enabled or disabled

`VDI_FAILURE`
handle contains invalid values

EXAMPLE see section 5.4.25 "vdi_smem_map(), vdi_smem_unmap()" on page 149

5.4.27 vdi_transfer_set(), vdi_transfer_get()

SYNTAX
```
#include <sys/vdi.h>
int vdi_transfer_set( dev_info_t *dip, bt_t tm )
int vdi_transfer_get( dev_info_t *dip, bt_t *tm )
int vdi_transfer_free( dev_info_t *dip )
```

DESCRIPTION `vdi_transfer_set()`
controls in parts the setup of master windows for the specified driver. This function is the only way to pass extended information about master window properties to the VME nexus driver via the standard Solaris mapping call `ddi_map_regs(9f)`.

Only the following subset of bus property literals can be used by this function: `VME_BT_PF`, `~_PRIAUTO`, `~_PROGAUTO`, `~_UNALIGN`, `~_WP` (see section 6.3 "Miscellaneous Bus Properties" on page 162). These are combined by the bit mask `VME_BT_TMASK`.

- The subset of available bus property literals may be restricted by hardware dependencies: not every hardware allows to set the bus properties on a per range basis.
- If a driver does not use `vdi_transfer_set()` and `VME:vme_master_defaults` is not used in `/etc/system` (see section 3.1 "Configuration" on page 17), the standard properties of the VME nexus driver will be used.
- If `vdi_transfer_set`() is used, the properties defined when calling `vdi_transfer_set`() will be used for all VME master windows that will be set up via `ddi_map_regs(9f)` or

2049369 -- 0 January 2000

`vdi_map()` for the specified driver. Existing mappings are not affected.

`vdi_transfer_get()`
returns the mode reservations for the specified device.

`vdi_transfer_free()`
deletes the transfer mode reservation for the specified driver. This means that the VME nexus driver's default values will be used when master windows are set up for this driver using `ddi_map_regs`(9f) or `vdi_map`().

VARIABLES

`dip`
device's information pointer: specifies the driver

`tm`
transfer modes to set. Only the bits masked by `VME_BT_TMASK` are used (see `sys/vme_types.h`).

RETURN VALUES

`vdi_transfer_get()` returns `VDI_SUCCESS` if a transfer mode was set for the specified driver or `VDI_FAILURE` if not.

`vdi_transfer_free()` always returns `VDI_SUCCESS`.

`vdi_transfer_set()` returns `VDI_SUCCESS` if successful. It returns `VDI_FAILURE` if the maximum number of transfer modes is exceeded or no kernel memory can be allocated.

EXAMPLE

```
{
  bt_t bt = VME_BT_WP;
  ...
  /* Enable write posting for all future master windows,
   * disable all other transfer modes
   */
  if (VDI_SUCCESS != vdi_transfer_set( vmeplusdip, bt ))
  {
          … /* ERROR */
  }
  ...
  if ( VDI_SUCCESS != vdi_transfer_get( vmeplusdip, &bt ) )
  {
          … /* ERROR */
  }
  ...
  /* Use the Nexus' default settings for future master
   * windows
   */
  (void)vdi_transfer_free( vmeplusdip ) )

  ...
}
```

5.4.28 vdi_virq_trigger(), vdi_virq_ackwait()

SYNTAX

```
#include <sys/vdi.h>
int vdi_virq_trigger(int level, int vector,
                          long timeout, u_int flags);
int vdi_virq_ackwait (int level, long timeout, u_int flags);
```

DESCRIPTION

As of Solaris VMEbus Driver release 2.1, assertion of VMEbus interrupts and specifying a time-out for the IACK cycle is supported in general by the software package.

Note: **Not all CPU boards support this feature. Refer to the *Release Notes* for limitations and hardware dependencies.**

If the requested time expires, an error is reported. This may be necessary for error recovery if the VMEbus interrupt handler does not acknowledge interrupts as it is expected to do.
The timer resolution is in system ticks, e.g. it is 10 ms in Solaris 2.5.

`vdi_virq_trigger()`
triggers an interrupt and waits until the acknowledge cycle has finished.

`vdi_virq_ackwait()`
waits until the most recent interrupt request has been acknowledged. This is not necessary if `vdi_virq_trigger()` has successfully acknowledged the interrupt already.

VARIABLES

`level`
interrupt level to be triggered

`vector`
interrupt vector to be triggered (only for `vdi_virq_trigger()`)

`timeout`
is a value in clock ticks that specifies the maximum time to wait for the IACK cycle to complete.

`flags`
is a bit mask which can be used to control various properties. The following bits are defined:

- `VIACK_DONTWAIT`
  If set to 1, the function returns immediately without waiting for the IACK cycle to finish. This flag is mutually exclusive with the `VIACK_ENDLESS` flag. Setting `VIACK_DONTWAIT` makes the `timeout` parameter obsolete.
- `VIACK_ENDLESS`
  If set to 1, the function does not return until the IACK cycle has

204936 9 -- 0 January 2000

been completed. This flag is mutually exclusive with the VIACK_DONTWAIT flag. Setting this flags makes the timeout parameter obsolete.

– VIACK_INTERRUPTIBLE
If set to 1, the function may be interrupted by a signal while waiting for an interrupt acknowledge. If set to 0, it is not interruptible.

– P_VIACK_DONTWAIT (vdi_virq_trigger() only)
If some thread is still waiting for an IACK to be completed at the time vdi_virq_trigger() is called, the calling thread might block until the other thread either completes the IACK successfully or decides to stop waiting. If it decides to stop waiting, the behavior of vdi_virq_trigger() depends on the P_VIACK_DONTWAIT flag.

If P_VIACK_DONTWAIT is cleared, vdi_virq_trigger() waits for the old IACK to be completed with the given wait criteria. If a timeout occurs, the function fails with return value VDI_BUSY. If no timeout occurs (meaning that the previous IACK has finally completed in time), it will trigger the requested interrupt and use the timing criteria again to wait for its own IACK cycle to complete. If the timing citeria are violated this time, vdi_virq_trigger() will fail with return value VDI_TIMEOUT.

If P_VIACK_DONTWAIT is set, vdi_virq_trigger() fails immediately with return value VDI_BUSY if it detects that a pending IACK prevents the interrupt to be triggered and the originator has stopped waiting for it.

RETURN VALUES

VDI_SUCCESS
the interrupt was triggered by vdi_virq_trigger() and the IACK cycle completed successfully (unless VIACK_DONTWAIT was set in the request structure; in this case, the function returns VDI_SUCCESS immediately after successfully asserting the interrupt).

VDI_BUSY
The last IACK cycle on this level has not been finished yet. Note that it is not possible to remove an interrupt request that has not been acknowledged yet because this is forbidden by the VMEbus specification (see also the description for P_VIACK_DONTWAIT above).

VDI_TIMEOUT
The IACK cycle for the given request did not complete in the requested amount of time.

VDI_FAILURE
invalid parameters

204936 9 -- 0 January 2000

6 VME Bus Properties

When dealing with VMEbus accesses, be it master/slave windows, mailboxes, DMA transfers, or others, it is necessary to specify the properties of the VMEbus transaction (like for example address space or data).

Solaris bus types

Traditionally, Solaris defines the “bus types” mentioned above in `/usr/include/sys/bustypes.h`. All Solaris bus types are supported within the VME driver. However, the definitions do not cover all aspects and features provided by Force Computers’ VMEbus drivers.

Extended bus type concept: bus property

To cover all aspects and features provided by Force Computers’ VMEbus drivers, an extension of the bus type concept has been designed for use with the application and driver interfaces described in this manual. To easily distinguish the 2 concepts, the Force Computers extension uses the term “bus property” instead of “bus type”.

A bus property as supported by Force Computers’ VMEbus drivers

- is a bit set of type `bt_t`,
- with bits denoted by the `VME_BT_`*`prop`* macros described in this section and defined in `sys/vme_types.h`.

Global and per-range bus properties

To allow access to the VMEbus, the driver has to set up master windows. Depending on the hardware, some properties can be applied to each master window individually (for example the VMEbus address space), others only globally to all windows. The first are referred to as “per range” bus properties, the latter as “global” bus properties.
The same applies to slave windows, which are needed to operate the board in slave mode.

6.1 Address Spaces – VME_BT_A*xx* and VME_BT_CRCSR

The VME_BT_A*xx* and VME_BT_CRCSR literals define the address space where a data transfer takes place.

- Interpretation for master windows: VME_BT_A*xx* literals specify the address space used for VMEbus master accesses. A master window has one and only one address space property.
- Interpretation for slave windows: A slave window can have one or more of the VME_BT_A*xx* bus properties, meaning that it will accept transfers in each of them. The same is true for mailboxes, which can reside in multiple address spaces as well.

VME_BT_A16	A16 address space (AM codes 29_{16} and $2D_{16}$).
VME_BT_A24	A24 address space (AM codes 38_{16} including $3F_{16}$)
VME_BT_A32	A32 address space (AM codes 08_{16} including $0F_{16}$)
VME_BT_A40	A40 address space (future extension)
VME_BT_A64	A64 address space (future extension)
VME_BT_CRCSR	Allows access to the CR/CSR address space
VME_BT_AMASK	A bit set combining all VME_BT_A*xx* and VME_BT_CRCSR literals.

2048369 – 0 January 2000

6.2 Data Modes – VME_BT_D*xx*

The VME_BT_D*xx* literals describe the way data is transferred, i.e. the data width or the kind of burst (block) transfer.

- Interpretation for master windows: A master window can be marked with any combination of VME_BT_D*xx* literals, each one denoting an access width from the host bus (e.g. SBus) that can be transformed into the corresponding access width on the VMEbus. This is also true for local bus burst transfers, which may cause a VMEbus block transfer if the corresponding bus property bit is set.

  If the VME_BT_DAUTO property is set, the VMEbus interface hardware will resize accesses of widths which do not appear as VME_BT_D*xx* literal into accesses that are possible. Example:

  > A master window with data bus properties VME_BT_D8|VME_BT_DAUTO resizes all kinds of local bus data widths to 8 bit accesses on the VMEbus.

- Interpretation for slave windows and mailboxes: The data bus property literals denote the VMEbus access widths which are decoded by the slave window to some kind of local bus master access or which cause a mailbox interrupt, respectively.

VME_BT_D8	8 data bits, single cycle transfer.
VME_BT_D16	16 data bits, single cycle transfer.
VME_BT_D32	32 data bits, single cycle transfer.
VME_BT_D64	64 data bits, single cycle transfer (future extension).
VME_BT_DAUTO	Automatic data resize. This is only meaningful for master windows.
VME_BT_BLT	Block transfer.
VME_BT_MBLT	Multiplexed block transfer.
VME_BT_2EVME	2eVME transfer.
VME_BT_DMASK	A bit set combining all literals mentioned above (VME_BT_D8, …, VME_BT_2EVME).

6.3 Miscellaneous Bus Properties

The literals described in this section define miscellaneous VMEbus bus properties.

- The interpretation for master windows depends on other bus property bits being set:
  - If the “write posting” bus property bit is set for a master window, the VMEbus interface hardware will acknowledge write transactions to the VMEbus immediately without waiting for the access to finish. This typically increases the transfer speed, but may cause problems when an error conditions occurs.
  - If the “data prefetch” bus property bit is set, the VMEbus interface hardware will perform “read ahead” on the VMEbus.
  - If the `VME_BT_NPRV` bus property bit is set, non-privileged VMEbus AM codes are generated. If it is cleared, privileged VMEbus AM codes are generated.
  - If the `VME_BT_PROG` bus property bit is set, program AM codes are generated. If it is cleared, data AM codes are generated respectively.
  - There are 2 programmable AM codes available for example for FGA-5100 based CPU boards. `VME_BT_PAMC`*x* literals specify the address space of the programmable AM code used for VMEbus master accesses.
- The interpretation for slave windows depends on other bus property bits being set:
  - If the “write posting” bus property bit is set for a slave window, VMEbus write accesses to the slave window will be acknowledged immediately to the originating master without waiting for the transaction to finish. This typically increases the transfer speed, but causes problems when an error conditions occurs.
  - If the “data prefetch” bus property bit is set for a slave window, the VMEbus interface hardware will perform “read ahead” on the host bus.
  - If the `VME_BT_NPRV` bus property bit is set, both privileged and non-privileged VMEbus AM codes will be accepted. If it is cleared, only privileged AM codes will be accepted.
  - If the `VME_BT_PROG` bus property bit is set, both data and program VMEbus AM codes will be accepted. If it is cleared, only data AM codes will be accepted.

`VME_BT_MMASK` A bit mask combining all miscellaneous bus properties.

204936 9 – 0 January 2000

`VME_BT_PAMASK` A bit set combining all `VME_BT_PA`*`M`*`C`*`x`* literals.

`VME_BT_PAMC1` First programmable AM code.

= `1` selected

= `0` de-selected

`VME_BT_PAMC2` Second programmable AM code.

= `1` selected

= `0` de-selected

`VME_BT_PF` Data read prefetch enable.

= `1` enable

= `0` disable

`VME_BT_PRIAUTO` Automatic privileged/non-privileged AM code generation, depending on the state of the processor. Only valid for master windows.

= `1` generate supervisory or user AM codes depending on the mode the processor is currently running in

= `0` disable this feature

`VME_BT_PROG` Program access AM code enable.

= `1` generate program AM codes

= `0` generate data AM codes

`VME_BT_PROGAUTO` Automatic program/data AM code generation depending on the type of access the processor does. Only valid for master windows.

= `1` generate program or data AM codes depending on the type of access

= `0` disable this feature

`VME_BT_NPRV` Non-privileged AM code enable.

Compatibility note `VME_BT_NPRV`

Note: **The name of `VME_BT_NPRV` used to be `VME_BT_USER` in previous releases. Both names can be used though `VME_BT_NPRV` is preferred.**

= `1` generate non-privileged (user) AM codes

= `0` generate privileged (supervisory) AM codes

`VME_BT_TMASK` A bit mask combining all bus properties which can be used in combination with the `vui_transfer_mode_set/~_get` functions (see page 40) and the `vdi_transfer_set/~_get` functions (see

page 154): VME_BT_PF, ~_PRIAUTO, ~_PROG, ~_PROGAUTO, ~_UNALIGN, ~_USER, ~_WP.

VME_BT_UNALIGN Unaligned accesses possible

= 1 allow unaligned accesses

= 0 prohibit unaligned accesses

VME_BT_USER See “Compatibility note VME_BT_NPRV” on page 163.

VME_BT_WP Write posting enable.

= 1 enable

= 0 disable

VME_BT_RESERVED Reserved bit.

204936 9 – 0 January 2000

7 System Messages

This section lists the drivers' system messages and documents possible causes.

7.1 Panic Messages

`PANIC: SBus virt. address: xx (no IOMMU mapping?)`
A VMEbus master accessed the local CPU's slave window and an error occurred. The error was not acknowledged to the master because the slave window was marked `write posted`. For information on changing the system's behavior, see section 5.3 "Nexus Driver Fault Handling" on page 107.

`PANIC: no sbus node`
The `sbus` node is missing in the device tree.

`WARNING: S-to-VME Write Posting error at vme-xx`
`PANIC: panic`
A VMEbus write access performed via a master window marked as `write posted` resulted in a bus error. For information on changing the system's behavior, see section 5.3 "Nexus Driver Fault Handling" on page 107.

`PANIC: VME=xxx BT=xxx`
A VMEbus write access resulted in a bus error. For information on changing the system's behavior, see section 5.3 "Nexus Driver Fault Handling" on page 107.

7.2 Warnings

Errors on the VMEbus

VMEbus transactions which terminate with a BERR are usually logged on the console. The information printed consists of

- the VMEbus address where the fault occurred, and
- a “bus property” bit set (“BT=” or “BP=”), which basically encodes the VME AM code (see section 6 “VME Bus Properties” on page 159).

Beneath the VMEbus related information, there might be additional messages visible which show the errors reported by the underlying bus nexus driver.

```
WARNING: Async Fault from S-to-VME (superv.)
WARNING: VME=xx BT=yy
```

Caused by a kernel access an asynchronous write error occurred on the VMEbus. The behavior on write errors can be controlled via the `/etc/system` file (see section 5.3 “Nexus Driver Fault Handling” on page 107).

```
WARNING: Async Fault from S-to-VME (non-priv)
WARNING: VME=xx BT=yy
```

Caused by a process access via `mmap()` an asynchronous write error occurred on the VMEbus. The behavior on write errors can be controlled via the `/etc/system` file (see section 5.3 “Nexus Driver Fault Handling” on page 107).

```
WARNING: VME synchronous error: ctx=xx VME=yy BT=zz
```

Caused by a kernel access a read error occurred on the VMEbus. The behavior on kernel read errors can be controlled via the `/etc/system` file (see section 5.3 “Nexus Driver Fault Handling” on page 107).

```
WARNING: L-to-VME Write Posting error at vme xxx
```

A write posted VMEbus master access caused an error and the originator of the access (the process) could not be determined. The behavior on write posted errors can be controlled via the `/etc/system` file (see section 5.3 “Nexus Driver Fault Handling” on page 107).

```
WARNING: (VME): VME Slave to Local Bus posted write error
```

A VMEbus master accessed the local CPU via a slave window which was set up as write posted and an error occurred. The behavior on write posted slave access errors can be controlled via the `/etc/system` file (see section 5.3 “Nexus Driver Fault Handling” on page 107).

```
WARNING: vmectl: could not send sig. xxx to process yyy
```

A process has set up a signal for a VMEbus event but did not release it properly. The event occurred and the driver detected that the process that owned the event does not exist any more. This message will appear only once, the driver releases events after this situation has occurred.

2049369 – 0 January 2000

`WARNING: vector xxx handles more than one VME IRQ`
One interrupt vector is defined for several interrupt levels. Make sure that this is what you intended.

`WARNING: spurious VMEbus interrupt on level xx, vec yy`
An interrupt acknowledge cycle resulted in an interrupt vector for which no interrupt handler was defined. Make sure you specified the correct interrupt vector for your hardware in the *driver*`.conf` file.

`WARNING: vme: no interrupt vector for VME IRQ xx`
A device has triggered an interrupt but didn't provide an interrupt vector.

`WARNING: vme_attach: vdi_init failed`
or
`WARNING: vdi_init: vhi_init failed`
Proper hardware was detected by the VME nexus driver but it could not be initialized.

`WARNING: VME: VME DMA not possible, set the slavewin property in VME.conf`
Some device driver tried to initiate a DMA transfer to the local CPU by means of the DDI DMA interface. To do so, it is necessary to set the `slavewin` property in the configuration file of the VMEbus nexus driver (see section 5.1.2 "Slave Window Property" on page 103).

`WARNING: VME DMA ERR on VMEbus addr xx`
A DMA transfer aborted because a bus error on the VMEbus occurred.

`WARNING: VME DMA ERR on local addr xx`
A DMA transfer aborted because a bus error on the host bus (e.g. SBus) occurred.

`NOTICE: vmeplus_segmap: No reg property`
A mapping request failed because there was no `reg` property in the configuration file of `vmeplus`. Normally, it should not be necessary to touch the `reg` property entries in this file.

`WARNING: map_slave(): cannot allocate fdma space for len xxx`
The fast DMA driver failed to allocate a DMA buffer of the requested size.

`WARNING: vmeplus_intr: ringbuffer is full, dropped interrupt <n>`
The vmeplus driver uses a ring-buffer to process information for high level interrupts. If the driver gets more interrupts than can be handled, the ring buffer may overflow and interrupt events get lost. The ring buffer size may be incremented by setting the `vmeplus:rb_size` variable in `/etc/system` appropriately.

`WARNING: vme0: <driver>: VME level <n> in use or grabbed`
A device driver has made an attempt

– to either grab the specified VMEbus interrupt level, but this level is already used by another interrupt handler,

– or to set up a vectored interrupt handler for the specified level, but this level is already grabbed by another interrupt handler.

7.3 Notices

`NOTICE: VME fault handling is OFF!`
This message occurs if the `VME:vme_fault_hndl_off` flag has been set in `/etc/system`. Errors on the VMEbus will most likely cause the system to panic.

`NOTICE: vdi_smem_map: SMEM_FIXED not supported`
or
`NOTICE: vdi_smem_map: PADDR not supported`
Allocation of slave memory or DMA buffer space failed because the `flags` parameter of the request contains an entry that is currently not supported on the underlying architecture.

2049369 – 0 January 2000

Product Error Report

PRODUCT:	SERIAL NO.:
DATE OF PURCHASE:	ORIGINATOR:
COMPANY:	POINT OF CONTACT:
TEL.:	EXT.:
ADDRESS: _____ _____ _____	
PRESENT DATE:	
AFFECTED PRODUCT: ❑ HARDWARE ❑ SOFTWARE ❑ SYSTEMS	AFFECTED DOCUMENTATION: ❑ HARDWARE ❑ SOFTWARE ❑ SYSTEMS
ERROR DESCRIPTION: _____ _____ _____ _____ _____ _____ _____ _____ _____ _____	

THIS AREA TO BE COMPLETED BY FORCE COMPUTERS:

DATE:

PR#:

RESPONSIBLE DEPT.: ❑ MARKETING ❑ PRODUCTION
ENGINEERING ➠ ❑ BOARD ❑ SYSTEMS

✉ Send this report to the nearest Force Computers headquarter listed on the back of the title page.